Using PagerDuty Terraform Provider

show me how to use the pagerduty terraform provider to connect to a pagerduty instance and define a service

---

Bot

To get started with the PagerDuty Terraform provider, you'll need to configure the provider block with your API token and then define resources like pagerduty_service.

Prerequisites

Before writing the Terraform configuration, ensure you have a PagerDuty API token. You can generate this in the PagerDuty web UI under **Integrations > API Access Keys**.

Provider Configuration

First, you need to tell Terraform to use the PagerDuty provider and authenticate it. You can do this by setting the token argument in the provider block. While you can hardcode the token, it is safer to pass it as a variable.

```
terraform {
  required_providers {
    pagerduty = {
      source  = "PagerDuty/pagerduty"
      version = "~> 2.0"
    }
  }
}

variable "pagerduty_token" {
  type        = string
  description = "Your PagerDuty API Token"
  sensitive   = true
}

provider "pagerduty" {
  token = var.pagerduty_token
}
```

Defining an Escalation Policy

A PagerDuty service usually requires an Escalation Policy to know who to alert. If you don't have an existing one you want to reference, you can create a simple one.

```
resource "pagerduty_user" "example_user" {
  name  = "Jane Doe"
  email = "jane.doe@example.com"
}

resource "pagerduty_escalation_policy" "example_policy" {
  name      = "Engineering Escalation Policy"
  num_loops = 2

  rule {
    escalation_delay_in_minutes = 10
    target {
      type = "user_reference"
      id   = pagerduty_user.example_user.id
    }
  }
}
```

Defining the Service

Now you can define the pagerduty_service. This resource connects the escalation policy to the service settings, such as the auto-resolve timeout and acknowledgement timeout.

```
resource "pagerduty_service" "example_service" {
  name                    = "My Web App Service"
  auto_resolve_timeout    = 14400

# 4 hours
  acknowledgement_timeout = 600

# 10 minutes
  escalation_policy       = pagerduty_escalation_policy.example_policy.id
  alert_creation          = "create_alerts_and_incidents"
}
```

Applying the Configuration

Once you have your files saved (e.g., main.tf), you can initialize and apply the configuration. When asked for the variable, paste your API token.

```
terraform init
terraform apply
```

---

can you give me a downladable main.tf for the example above

Bot

I can certainly set that up for you. Here is a Python script that will generate the main.tf file containing the complete configuration we just discussed, including the provider setup, user, escalation policy, and service definition.

```
tf_content = """terraform {
  required_providers {
    pagerduty = {
      source  = "PagerDuty/pagerduty"
      version = "~> 2.0"
    }
  }
}

variable "pagerduty_token" {
  type        = string
  description = "Your PagerDuty API Token"
  sensitive   = true
}

provider "pagerduty" {
  token = var.pagerduty_token
}

# Create a user to assign to the policy
resource "pagerduty_user" "example_user" {
  name  = "Jane Doe"
  email = "jane.doe@example.com"
# Change this to a valid email in your org
}

# Create an Escalation Policy
resource "pagerduty_escalation_policy" "example_policy" {
  name      = "Engineering Escalation Policy"
  num_loops = 2

  rule {
    escalation_delay_in_minutes = 10
    target {
      type = "user_reference"
      id   = pagerduty_user.example_user.id
    }
  }
}

# Create the Service using the Escalation Policy
resource "pagerduty_service" "example_service" {
  name                    = "My Web App Service"
  auto_resolve_timeout    = 14400

# 4 hours
  acknowledgement_timeout = 600

# 10 minutes
  escalation_policy       = pagerduty_escalation_policy.example_policy.id
  alert_creation          = "create_alerts_and_incidents"
}
"""

with open("main.tf", "w") as f:
    f.write(tf_content)

# Created/Modified files during execution:
print("main.tf")
```

---

how do I use the output above?

Bot

To use the main.tf file generated above, you will need to have Terraform installed on your machine and a valid PagerDuty API token.

1. Save the File

If you ran the python code, the file main.tf is created in your current directory. If you just copied the text, create a new folder on your computer, open a text editor (like VS Code, Notepad++, or Sublime Text), paste the code into a new file, and save it as main.tf.

2. Initialize Terraform

Open your terminal or command prompt and navigate to the folder where you saved main.tf. Run the following command to download the necessary PagerDuty provider plugins:

```
terraform init
```

3. Plan the Deployment

Run a "plan" to see what Terraform intends to build. This acts as a dry run. Because the configuration uses a variable for the token, you have two ways to pass it:

Option A: Interactive Input
Run the command below. Terraform will pause and ask you to paste your token.

```
terraform plan
```

Option B: Environment Variable (Recommended)
You can set an environment variable prefixed with TF_VAR_ to avoid typing it every time.

```
# Mac/Linux
export TF_VAR_pagerduty_token="your-actual-api-token-here"

# Windows PowerShell
$env:TF_VAR_pagerduty_token="your-actual-api-token-here"

terraform plan
```

4. Apply the Configuration

If the plan looks correct (it should say it will create 3 resources), apply the changes to your actual PagerDuty account:

```
terraform apply
```

Type yes when prompted to confirm.

5. Verify in PagerDuty

Once the apply is complete, log in to your PagerDuty console. You should see:

- A new User named "Jane Doe".
- A new Escalation Policy named "Engineering Escalation Policy".
- A new Service named "My Web App Service".

---

how do I run the python code on macos

Bot

To run the Python code on macOS, you'll need to use the
Terminal application. Here is a step-by-step guide to getting that main.tf file created.

1. Open Terminal

You can find this by pressing Command + Space to open Spotlight Search, typing "Terminal", and pressing Enter.

2. Create the Python Script

You need to put the Python code into a file. You can do this easily using a text editor like nano right inside the terminal.

1. Type nano generate_tf.py and press Enter. (This opens a simple text editor).
2. Paste the Python code I gave you earlier into this window.
3. Press Control + O (Letter O) then Enter to save the file.
4. Press Control + X to exit the editor.

3. Run the Script

Now that you have the file saved, you can run it using the python command. macOS usually comes with Python 3 installed.

Type the following command and press Enter:

```
python3 generate_tf.py
```

(Note: If your system just uses python instead of python3, try that if the above doesn't work, but python3 is standard on modern macOS).

4. Verify the File Creation

If the script ran successfully, you won't see any errors. You can check if the main.tf file was created by listing the files in the directory:

```
ls -l main.tf
```

You should see main.tf listed there. You can now proceed with the Terraform commands (terraform init, etc.) in that same directory.

Okay, I got most of that working, what I now need to do is pass the api token to terraformer when I run the command: terraformer import pagerduty --resources="*"

Bot

It looks like you are switching gears from *Terraform* (creating infrastructure) to *Terraformer* (a tool by Google Cloud capable of importing existing infrastructure *into* Terraform code).

Terraformer works slightly differently than the standard Terraform CLI. Instead of looking for TF_VAR_ environment variables or variables defined in a .tf file, Terraformer typically looks for specific environment variables that match the provider's standard configuration, or it allows you to pass arguments directly.

For the PagerDuty provider in Terraformer, the standard way to pass credentials is via an **environment variable**.

The Standard Method (Environment Variable)

The underlying Terraform provider for PagerDuty looks for the environment variable PAGERDUTY_TOKEN. Setting this before running the command is the most reliable method.

Run this in your terminal:

```
# 1. Set the variable
export PAGERDUTY_TOKEN="your-actual-api-token-here"

# 2. Run the import
terraformer import pagerduty --resources="*"
```

Alternative Method (Configuration File)

If the environment variable doesn't work (which can happen depending on the version of Terraformer or the specific provider build), Terraformer allows you to pass provider configuration in an init.tf file within your working directory, but it's often easier to just pass the arguments directly if supported.

However, almost all PagerDuty/Terraform interactions default to PAGERDUTY_TOKEN.

Important Note on --resources="*"

Using wildcard * for resources can sometimes be buggy or extremely slow if you have a massive PagerDuty account. If the command fails or hangs, try importing specific resources instead, like this:

```
terraformer import pagerduty --resources="users,services,escalation_policies"
```

Summary of Flags

Just in case you run into version-specific syntax issues, verify you have the basics covered:

1. **--resources**: What you want to download (e.g., * or users,services).
2. **--connect**: (Optional) Defaults to true, but ensures Terraformer connects to the provider.
3. **Authentication**: Provided via export PAGERDUTY_TOKEN=....

---

do any of the Data Sources available using the pagerduty terraform provider expose service dependencies?

Bot

Yes, you can access service dependencies using the PagerDuty Terraform provider, but there is a nuance: **Service Dependencies are not exposed as a standalone Data Source** in the same way a single User or Service is.

Instead, dependencies are typically managed via specific resources. However, if you are looking to *read* existing dependencies via a Data Source, you often have to look at the **Business Service** data source, as dependencies are usually defined in the context of a Business Service dependent on Technical Services.

1. pagerduty_business_service Data Source

You can retrieve a business service and its metadata. While the main data "pagerduty_service" generally retrieves Technical Services, the Business Service is where dependencies are most commonly visualized in the PagerDuty Graph.

```
data "pagerduty_business_service" "example" {
  name = "My Critical Business Flow"
}
```

Note: This data source itself returns the ID and name, but it doesn't always deeply list all downstream dependencies in a simple list attribute within the data source outputs.

2. The Relationship Resource: pagerduty_service_dependency

If you are trying to *manage* or *import* dependencies, you are looking for the resource pagerduty_service_dependency.

Unfortunately, **there is currently no specific data "pagerduty_service_dependency"** that lets you query "What services does Service A depend on?" directly to get a list back for dynamic use in other Terraform logic.

How to work around this

If you need to get the dependencies to use them elsewhere in your Terraform code, you usually have to rely on the fact that you know the IDs of the services involved.

If you are trying to **import** existing dependencies so you can manage them in code (which matches your earlier questions about terraformer), you would import them as pagerduty_service_dependency resources.

The structure looks like this:

```
resource "pagerduty_service_dependency" "example" {
  dependency {
    measure_service   = pagerduty_business_service.marketing.id
    measure_dependent = pagerduty_service.user_db.id
    type              = "business_service"
  }
}
```

Summary of the limitation:

- **Can execute:** You *can* create dependencies using the resource block.

- **Cannot execute:** You *cannot* easily query the API via a simple data block to ask "Give me a list of all dependencies for Service X" inside Terraform code. You generally have to know the IDs of the two services you want to link.

---

so, is there a convenient way I could get a list of all the business services using terraformer and then use that list to retrieve the service dependencies for each business service using the pagerduty terraform provider?

Bot

The short answer is **no**, you cannot perform that specific workflow (Get List A -> Loop through List A to fetch Data B) purely using Terraform Data Sources.

Terraform Data Sources are not designed to be used as a recursive discovery tool. Even if you get a list of Business Services, the pagerduty_business_service data source **does not expose a list of its dependencies as an output attribute**. Without that attribute, you cannot iterate (for_each) over them to find the connected services.

However, **Terraformer** can solve this without the two-step process.

The Solution: Import service_dependencies Directly

You do not need to fetch the Business Services first to find the dependencies. You can tell Terraformer to scrape the entire account for **all** service dependencies directly.

The PagerDuty API treats a "Service Dependency" as its own distinct object/resource, separate from the Service itself. Therefore, you can import them in one go.

Run this command:

```
export PAGERDUTY_TOKEN="your-api-token"

terraformer import pagerduty --resources="business_services,service_dependencies"
```

What this will do

1. **business_services**: It will generate a resource block for every Business Service you have.

2. **service_dependencies**: It will generate a pagerduty_service_dependency resource block for every connection between services.

The Output You Will Get

Terraformer will generate a .tf file (likely generated/pagerduty/service_dependencies/service_dependency.tf) that looks like this:

```
resource "pagerduty_service_dependency" "tfer--dependency_ID_001" {
    dependency {
        measure_service   = "PXXXXX"

# The ID of the Business Service
        measure_dependent = "PYYYYY"

# The ID of the Technical Service
        type              = "business_service"
    }
}
```

This effectively gives you the map you are looking for. You can look at measure_service (the parent) and measure_dependent (the child) to see how everything connects.

Why the "Data Source" method you proposed won't work

If you tried to write Terraform code to do this dynamically, you would hit a wall:

1. data "pagerduty_business_service" inputs a Name/ID and outputs configuration (Teams, Owner), but **not** a list of dependent service IDs.
2. Because the Parent doesn't output the Children's IDs, you have nothing to pass into a loop to look up the next level.

Using terraformer to bulk-import the service_dependencies resource type is the only way to "discover" these relationships without writing a custom Python script against the API.

```
terraformer import pagerduty --resources="business_services,service_dependencies"
2025/12/18 15:55:29 pagerduty importing... business_services
2025/12/18 15:55:29 pagerduty error importing business_services, err: pagerduty: business_services not supported service
```

```
2025/12/18 15:55:29 pagerduty importing... service_dependencies
2025/12/18 15:55:29 pagerduty error importing service_dependencies, err: pagerduty: service_dependencies not supported service
2025/12/18 15:55:29 pagerduty Connecting....
```

Bot

It seems terraformer does not support business_services or service_dependencies for PagerDuty, even though those resources exist in the Terraform *Provider*.

The error not supported service usually comes from Terraformer's internal mapping, not the PagerDuty API itself. Terraformer maintains a hardcoded list of resources it knows how to import, and it appears these newer PagerDuty features haven't been added to that list yet.

Since you cannot use Terraformer for this specific task, you have to use a **Python script** to query the API directly and print out the dependencies.

Solution: Python Script to Map Dependencies

This script will hit the PagerDuty API, find all Business Services, and then find the Technical Services connected to them.

1. Requirements

You will need the requests library.

```
pip install requests
```

2. The Python Script (pd_deps.py)

Copy the code below. Replace YOUR_API_KEY at the top (or set it as an env var).

```
import os
import requests
import json
```

```
# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_all_items(endpoint, key_name, params=None):
    """Refetch pages until all items are collected."""
    if params is None:
        params = {}

    all_items = []
    params['limit'] = 100
    params['offset'] = 0
    more = True

    while more:
        response = requests.get(f"{BASE_URL}/{endpoint}", headers=headers,
params=params)
        if response.status_code != 200:
            print(f"Error fetching {endpoint}: {response.text}")
            break

        data = response.json()
        items = data.get(key_name, [])
        all_items.extend(items)

        more = data.get("more", False)
        params['offset'] += params['limit']

    return all_items

def main():
    print("Fetching Business Services...")
    # Fetch all Business Services
    biz_services = get_all_items("business_services", "business_services")

    print(f"Found {len(biz_services)} Business Services. Checking dependencies...\n")

    results = []

    for bs in biz_services:
        bs_id = bs['id']
        bs_name = bs['name'] or "Unnamed"
        # The Service Dependencies API relates things via 'relationships'
        # We query /service_dependencies to see what this BS depends on
        # You can filter by 'relationships[source_service_id]' or
'relationships[dependent_service_id]'

        # We want to know: "What technical services support this Business Service?"
        # So the Business Service is the "dependent" (it depends on tech services)
        # OR usually in PD 'Service Graph', the Business Service is the 'supporting' (parent)?
        # Actually: Structure is Business Service -> (depends on) -> Technical Service.
        # So we query where source_service_id = Business Service ID.

        params = {
            "relationships[source_service_id]": bs_id
            # "relationships[dependent_service_id]": ... if looking upstream
        }

        # Note: The endpoint is /service_dependencies
        deps = get_all_items("service_dependencies", "relationships", params=params)

        for dep in deps:
            # dependent_service is the one BEING depended on (The Technical Service)
            # source_service is the one doing the depending (The Business Service)

            support_svc = dep.get('dependent_service', {})
            support_id = support_svc.get('id')
            support_type = support_svc.get('type')

            results.append({
                "Business Service Name": bs_name,
                "Business Service ID": bs_id,
                "Depends On ID": support_id,
                "Depends On Type": support_type,
                "Relationship ID": dep.get('id')
            })

    # Output results
    print(f"{'Business Service':<30} | {'Dependency ID':<20} | {'Type'}")
    print("-" * 70)
    for r in results:
        print(f"{r['Business Service Name']:<30} | {r['Depends On ID']:<20} | {r['Depends On
Type']}")

    # Optional: Generate Terraform code
    with open("dependencies.tf", "w") as f:
        for r in results:
            tf_block = f"""
resource "pagerduty_service_dependency" "dep_{r['Relationship ID']}" {{
  dependency {{
    dependent_service {{
      id   = "{r['Depends On ID']}"
      type = "{r['Depends On Type']}"
    }}
    source_service {{
      id   = "{r['Business Service ID']}"
      type = "business_service"
    }}
  }}
}}
"""
            f.write(tf_block)

if __name__ == "__main__":
    main()
```

3. Run the script

```
export PAGERDUTY_TOKEN="your-actual-api-token"
python3 pd_deps.py
```

What this does

1. **Lists:** It prints a table to your console showing which Business Service relies on which Technical Service.
2. **Generates Terraform:** It creates a file named dependencies.tf containing the resource "pagerduty_service_dependency" blocks you were trying to import.

You can then run terraform init and terraform import (manually mapping them) OR simply add this .tf file to your configuration and run terraform apply (which might fail if they exist, requiring terraform import command generation, but at least you now have the resource definitions written out).

```
python3 ps_deps.py
Traceback (most recent call last):
  File "/Users/rlambie/.terraform.d/ps_deps.py", line 2, in
    import requests
ModuleNotFoundError: No module named 'requests'
```

Bot

It looks like your Python environment is missing the requests library, which is required to make the API calls. You can fix this easily by installing it.

Run this command in your terminal:

```
pip3 install requests
```

(If that command fails or says command not found, try pip install requests).

Once the installation finishes, try running your script again:

```
python3 ps_deps.py
```

```
pip install requests
zsh: command not found: pip
```

Bot

It looks like pip isn't linked directly in your path, which is common on newer macOS installations (they use python3 and pip3 specifically).

Try using the Python command itself to install it. This is the most reliable way:

```
python3 -m pip install requests
```

After that finishes, run your script again:

```
python3 ps_deps.py
```

```
python3 ps_deps.py
/Users/rlambie/Library/Python/3.9/lib/python/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
Fetching Business Services...
Found 0 Business Services. Checking dependencies...
```

```
Business Service            | Dependency ID       | Type
------------------------------------------------------------------
```

Bot

The output confirms that the script ran successfully!

Why did it find 0 results?

The script logged in, queried the API, and received a "Success" (200 OK) response from PagerDuty. The result was simply an empty list.

This usually happens because of the distinction between **Technical Services** and **Business Services** in PagerDuty:

1. **Technical Services (The Standard):** These are what you see under **Service Directory > Services**. (e.g., "Database", "API", "Frontend"). This is what most people use.
2. **Business Services (The Advanced Layer):** These are under **Service Directory > Business Services**. These are high-level concepts like "Checkout Flow" or "User Signup" that model business impact.

The previous script **only looked for Business Services**. If you haven't explicitly created those, the result is 0.

Updated Script: Check *Technical* Services

Here is an updated version of the script. This one searches your standard **Technical Services** (the ones you probably have) to see if *they* have any dependencies defined in the PagerDuty Service Graph.

Overwrite your ps_deps.py file with this code:

```
import os
import requests
import json

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_all_items(endpoint, key_name, params=None):
    if params is None:
        params = {}

    all_items = []
    params['limit'] = 100
    params['offset'] = 0
    more = True

    while more:
        response = requests.get(f"{BASE_URL}/{endpoint}", headers=headers,
params=params)

        if response.status_code != 200:
            print(f"Error fetching {endpoint}: Status {response.status_code}")
            try:
                print(response.json())
            except:
                print(response.text)
            break

        data = response.json()
        items = data.get(key_name, [])
        all_items.extend(items)

        more = data.get("more", False)
        params['offset'] += params['limit']

    return all_items

def main():
    # 1. Fetch Standard Technical Services
    print("Fetching Technical Services (Standard)...")
    tech_services = get_all_items("services", "services")
    print(f"Found {len(tech_services)} Technical Services.")

    # 2. Fetch Business Services (Just in case)
    print("Fetching Business Services...")
    biz_services = get_all_items("business_services", "business_services")
    print(f"Found {len(biz_services)} Business Services.")

    all_services = tech_services + biz_services

    print(f"\nScanning {len(all_services)} total services for dependencies relationships...")
    results = []

    for svc in all_services:
        svc_id = svc['id']
        svc_name = svc.get('name') or svc.get('summary') or "Unnamed"

        # Check what this service depends on
        params = {
            "relationships[source_service_id]": svc_id
        }

        deps = get_all_items("service_dependencies", "relationships", params=params)

        for dep in deps:
            support_svc = dep.get('dependent_service', {})
            support_id = support_svc.get('id')
            support_type = support_svc.get('type')

            # Try to resolve friendlier name for the dependent service if we have it locally
            found = next((s for s in all_services if s['id'] == support_id), None)
            support_name = found.get('name', 'Unknown ID') if found else support_id

            results.append({
                "Source Name": svc_name,
                "Source ID": svc_id,
                "Depends On Name": support_name,
                "Depends On ID": support_id,
                "Relationship ID": dep.get('id')
            })

    # Output results
    if len(results) == 0:
        print("\nNo dependencies found! Your services might exist, but they are not linked in the
Service Graph.")
    else:
        print(f"\n{'Source Service':<30} | {'Depends On':<30}")
        print("-" * 65)
        for r in results:
            print(f"{r['Source Name']:<30} | {r['Depends On Name']:<30}")

        # Generate Terraform code
        print("\nGenerating dependencies.tf...")
        with open("dependencies.tf", "w") as f:
            for r in results:
                tf_block = f"""
resource "pagerduty_service_dependency" "dep_{r['Relationship ID']}" {{
  dependency {{
    dependent_service {{
      id   = "{r['Depends On ID']}"
      type = "service"
    }}
    source_service {{
      id   = "{r['Source ID']}"
      type = "service"
    }}
  }}
}}
"""
                f.write(tf_block)
        print("Done. Check dependencies.tf")

if __name__ == "__main__":
    main()
```

Run it again

```
python3 ps_deps.py
```

(Note on the SSL Warning): You can ignore the NotOpenSSLWarning. That is just a notice that the Python version on macOS is using LibreSSL instead of OpenSSL, but it is perfectly capable of making these secure requests.

```
Source Service                | Depends On
-----------------------------------------------------------------
Clone Club Web App Service    | Clone Club Web App Service
Default Service               | Clone Club Web App Service
OpenShift                     | Clone Club Web App Service
```

Bot

Excellent! The script successfully found your service dependencies.

At this point, you have a file named dependencies.tf in your folder that contains the Terraform code for these relationships.

However, do not run terraform apply just yet.

Since these dependencies **already exist** in PagerDuty, if you run terraform apply now, Terraform will try to create them again and fail with an "Already Exists" error. You need to **import** them into your Terraform state first.

How to Import these Dependencies

Because the Python script named the resources using their real PagerDuty IDs (e.g., dep_PABC123), the import process is straightforward.

1. **Open the file** dependencies.tf in your text editor.
2. Look at the resource names. They will look something like this:
   hcl
   ```
   resource "pagerduty_service_dependency" "dep_D123ABC" { ... }
   ```
3. **Run the import command** for each resource found.

You simply match the resource name to the ID at the end of the name.

For example, if the file contains dep_D123ABC, run:

```
terraform import pagerduty_service_dependency.dep_D123ABC D123ABC
```

Repeat this for the 3 dependencies found in your output.

Automatically Import (Optional Shortcut)

If you don't want to copy-paste manually, and you have grep and sed (standard on macOS), you can use this one-liner to parse the dependencies.tf file and generate the import commands for you:

```
grep "resource \"pagerduty_service_dependency\"" dependencies.tf | sed 's/resource
"pagerduty_service_dependency" "\(.*\)" {/terraform import pagerduty_service_dependency.
\1 \1/' | sed 's/dep_//2'
```

Note: The second sed command there handles the fact that the ID inside the command doesn't have the dep_ prefix, but the resource name does.

Or, simpler, just read the file and run the 3 commands manually!

Once Imported

After running the import commands:

1. Run terraform plan.
2. It should say **"No changes. Your infrastructure matches the configuration."**

You are now fully managing your Service Dependencies via Terraform

---

```
Fetching Technical Services (Standard)...
Found 3 Technical Services.
Fetching Business Services...
Found 1 Business Services.

Scanning 4 total services for dependencies relationships...

Source Service               | Depends On
------------------------------------------------------------
Clone Club Web App Service     | Clone Club Web App Service
Clone Club Web App Service     | Clone Club
Default Service              | Clone Club Web App Service
Default Service              | Clone Club
OpenShift                    | Clone Club Web App Service
OpenShift                    | Clone Club
Clone Club                   | Clone Club Web App Service
Clone Club                   | Clone Club
```

Not quite, the dependencies defined in pagerduty are:

"Clone club" is a business service that depends on "Clone Club Web App". "Clone club web App" is a technical service that depends on Openshift, which is also a technical service. "Default Service" is a technical service that has no dependencies.

Bot

It seems the previous script ran into issues either with how the IDs were being matched or confused the direction of the relationships (Parent vs. Child). The output showing a service dependent on itself is definitely incorrect.

To fix this and get a crystal clear picture of what is actually configured in PagerDuty (and ensure we generate the correct Terraform code), we should switch strategies.

Instead of looping through every service and asking "What do you depend on?", we will **download the entire Dependency Graph in one go** and map the names locally. This eliminates loop errors and ensures we see exactly what the API sees.

2. Improved Python Script

This version fetches **all** services and **all** dependencies separately and then joins them. It’s much more reliable.

Overwrite ps_deps.py with this:

```
import os
import requests
import json

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_all_items(endpoint, key_name, params=None):
    if params is None:
        params = {}

    all_items = []
    params['limit'] = 100
    params['offset'] = 0
    more = True

    while more:
        response = requests.get(f"{BASE_URL}/{endpoint}", headers=headers,
params=params)
        if response.status_code != 200:
            print(f"Error fetching {endpoint}: {response.status_code}")
            break

        data = response.json()
        items = data.get(key_name, [])
        all_items.extend(items)
        more = data.get("more", False)
        params['offset'] += params['limit']

    return all_items

def main():
    print("1. Loading all services (Technical & Business)...")

    # 1. Build a dictionary of ALL services { "ID": "Name" }
    service_map = {}

    tech_services = get_all_items("services", "services")
    for s in tech_services:
        service_map[s['id']] = s.get('name') or s.get('summary')

    biz_services = get_all_items("business_services", "business_services")
    for s in biz_services:
        service_map[s['id']] = s.get('name') or s.get('summary')

    print(f"   Mapped {len(service_map)} total services.\n")

    # 2. Fetch the Relationship Graph
    print("2. Fetching Service Dependencies...")
    relationships = get_all_items("service_dependencies", "relationships")

    results = []

    # 3. Process Relationships
    # In PagerDuty API:
    # "source_service" = The Parent (The one that HAS the dependency)
    # "dependent_service" = The Child (The one providing the functionality)

    for r in relationships:
        rel_id = r['id']

        src_obj = r.get('source_service', {})
        src_id = src_obj.get('id')
        src_type = src_obj.get('type', 'service')

# usually 'business_service' or 'service'

        dep_obj = r.get('dependent_service', {})
        dep_id = dep_obj.get('id')
        dep_type = dep_obj.get('type', 'service')

        # Resolve Names
        src_name = service_map.get(src_id, f"Unknown ID ({src_id})")
        dep_name = service_map.get(dep_id, f"Unknown ID ({dep_id})")

        results.append({
            "rel_id": rel_id,
            "src_name": src_name,
            "src_id": src_id,
            "src_type": src_type,
            "dep_name": dep_name,
            "dep_id": dep_id,
            "dep_type": dep_type
        })

    # 4. Output textual table
    if not results:
        print("No dependencies found in the PagerDuty graph.")
        return

    print(f"{'Parent (Depends On) ->':<35} | {'Child (Is Depended On)'}")
    print("-" * 70)

    for res in results:
        print(f"{res['src_name']:<35} | {res['dep_name']}")

    # 5. Generate Terraform
    print("\nGenerating dependencies.tf...")

    with open("dependencies.tf", "w") as f:
        for res in results:
            # Terraform block
            # Note: Terraform provider mimics API. source_service is the parent.
            # If "Clone Club" depends on "Web App", "Clone Club" is source.

            src_type_tf = "business_service" if "business" in res['src_type'] else "service"
            dep_type_tf = "business_service" if "business" in res['dep_type'] else "service"

            tf_block = f"""
resource "pagerduty_service_dependency" "dep_{res['rel_id']}" {{
  dependency {{
    dependent_service {{
      id   = "{res['dep_id']}"
      type = "{dep_type_tf}"
    }}
    source_service {{
      id   = "{res['src_id']}"
      type = "{src_type_tf}"
    }}
  }}
}}
"""
            f.write(tf_block)

    print("Done. You can now inspect 'dependencies.tf'.")

if __name__ == "__main__":
    main()
```

3. Run the new script

```
python3 ps_deps.py
```

4. Verify the Output

This script prints "Parent" vs "Child".

Based on your description, the output **should** look like this:

```
Parent (Depends On) ->             | Child (Is Depended On)
-----------------------------------------------------------------
Clone Club                      | Clone Club Web App
Clone Club Web App              | OpneShift
```

If the output is reversed (e.g., OpenShift | Clone Club Web App), it means the graph inside PagerDuty was actually configured backwards manually.

If the output is correct, the dependencies.tf file will now be accurate, and you can proceed with the import steps from the previous message (terraform import ...).

---

closer:

```
python3 ps_deps.py
/Users/rlambie/Library/Python/3.9/lib/python/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
1. Loading all services (Technical & Business)...
   Mapped 4 total services.

2. Fetching Service Dependencies...
Parent (Depends On) ->             | Child (Is Depended On)
-----------------------------------------------------------------
Unknown ID (None)                  | Clone Club Web App Service
Unknown ID (None)                  | Clone Club

Generating dependencies.tf...
Done. You can now inspect 'dependencies.tf'.
```

```
rlambie@rlambie-K9XQR37R0J .terraform.d % cat dependencies.tf

resource "pagerduty_service_dependency"
"dep_D2GM4S41FVBI9NQYPH7" {
  dependency {
    dependent_service {
      id   = "PA7HPI0"
      type = "service"
    }
    source_service {
      id   = "None"
      type = "service"
    }
  }
}

resource "pagerduty_service_dependency"
"dep_D2GM4TER8JNUZPCH11Z" {
  dependency {
    dependent_service {
      id   = "PJSUJGC"
      type = "business_service"
    }
    source_service {
      id   = "None"
      type = "service"
    }
  }
}
```

Bot

It looks like we are successfully talking to the API, but the **Source Service** (the parent) is coming back as None or missing from the specific API response payload we are getting.

This usually happens because the PagerDuty API returns slightly different data shapes depending on *how* you ask for the dependencies (whether you use the /service_dependencies endpoint directly or query via the Business Services endpoint).

When source_service is missing, it implies that the relationship object returned doesn't explicitly envelope the parent because of the context, or the API version requires us to look slightly differently.

The Fix: Debugging the Payload

Since source_service is returning None, we need to see exactly what PagerDuty is sending back so we can grab the rigth ID.

Here is a **Diagnostic Script**. This won't generate Terraform yet; it will just dump one raw relationship object so we can see where the ID is hiding.

1. Create debug_pd.py:

```
import os
import requests
import json

API_KEY = os.getenv("PAGERDUTY_TOKEN")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

print("Fetching one raw relationship object...")
resp = requests.get(f"{BASE_URL}/service_dependencies", headers=headers,
params={"limit": 1})

if resp.status_code == 200:
    data = resp.json()
    relationships = data.get('relationships', [])
    if relationships:
        print(json.dumps(relationships[0], indent=2))
    else:
        print("No relationships found.")
else:
    print(f"Error: {resp.status_code} - {resp.text}")
```

2. Run it:

```
python3 debug_pd.py
```

Anticipated Fix

I suspect the API is returning supporting_service vs dependent_service OR simpler objects.

However, if you want to skip the debug step and try the **most robust method** immediately, we can query relationships differently. Instead of asking for *all* relationships globally (which seems to be returning partial data), we will loop through your Business Service specifically, as that endpoint provides cleaner data about its own dependencies.

Try this final script (pd_deps_final.py). It iterates through Business Services to find their supports.

```
import os
import requests

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_items(endpoint, key, params=None):
    if not params: params = {}
    items = []
    params['limit'] = 100
    params['offset'] = 0
    more = True
    while more:
        r = requests.get(f"{BASE_URL}/{endpoint}", headers=headers, params=params)
        if r.status_code != 200: break
        d = r.json()
        items.extend(d.get(key, []))
        more = d.get('more', False)
        params['offset'] += params['limit']
    return items

def main():
    print("Fetching dependencies via Service traversal...")

    # 1. Get dictionary of Names for lookup
    svc_map = {}
    for s in get_items("services", "services"):
        svc_map[s['id']] = s.get('name')
    for s in get_items("business_services", "business_services"):
        svc_map[s['id']] = s.get('name')
    results = []

    # 2. Iterate ALL services (Business AND Technical) to see what supports them
    # We look at every service and ask "What supports you?"

    # Combined list
    all_ids = list(svc_map.keys())

    for svc_id in all_ids:
        # Get dependencies for this specific service
        # We ask: /service_dependencies?relationships[source_service_id]=SVC_ID
        # This returns the children (supporting services)

        rel_params = { "relationships[source_service_id]": svc_id }
        rels = get_items("service_dependencies", "relationships", rel_params)

        for r in rels:
            # When querying by source_service_id, the 'dependent_service' in the response is the child
            child = r.get('dependent_service', {})
            child_id = child.get('id')

            # The Parent is svc_id (because we queried for it)
            # The Child is child_id

            if child_id:
                results.append({
                    "rel_id": r['id'],
                    "parent_id": svc_id,
                    "parent_name": svc_map.get(svc_id, "Unknown"),
                    "parent_type": r.get('source_service', {}).get('type', 'service'),
                    "child_id": child_id,
                    "child_name": svc_map.get(child_id, "Unknown"),
                    "child_type": child.get('type', 'service')
                })

    # 3. Output
    print(f"\n{'Parent (Depends On)':<30} | {'Child (Is Depended On)'}")
    print("-" * 65)
    for res in results:
        print(f"{res['parent_name']:<30} | {res['child_name']}")

    # 4. Generate Terraform
    with open("dependencies.tf", "w") as f:
        for res in results:
            src_type = "business_service" if "business" in res['parent_type'] else "service"
            dep_type = "business_service" if "business" in res['child_type'] else "service"

            f.write(f"""
resource "pagerduty_service_dependency" "dep_{res['rel_id']}" {{
  dependency {{
    dependent_service {{
      id   = "{res['child_id']}"
      type = "{dep_type}"
    }}
    source_service {{
      id   = "{res['parent_id']}"
      type = "{src_type}"
    }}
  }}
}}
""")

if __name__ == "__main__":
    main()
```

```
Fetching dependencies via Service traversal...

Parent (Depends On)            | Child (Is Depended On)
----------------------------------------------------------------
Clone Club Web App Service     | Clone Club Web App Service
Clone Club Web App Service     | Clone Club
Default Service                | Clone Club Web App Service
Default Service                | Clone Club
OpenShift                      | Clone Club Web App Service
OpenShift                      | Clone Club
Clone Club                     | Clone Club Web App Service
Clone Club                     | Clone Club
```

Bot

It appears we are dealing with two specific issues:

1. **Filter Failure:** The PagerDuty API seems to be ignoring the filter in the loop, causing the "Cartesian product" result (where everything depends on everything).
2. **Missing Data:** In the global fetch, the source_service field appeared to be missing or named differently in the JSON response.

To solve this, we need to stop guessing the JSON structure and look at the raw data.

Please run this **Debug Script**. It will not generate Terraform code yet; instead, it will print the **RAW JSON** of one of your dependencies. This will show us exactly where the IDs are hiding so I can update the final script to match.

1. The Debug Script (debug_json.py)

```
import os
import requests
import json

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def main():
    print("Fetching global service dependencies to inspect JSON structure...")

    # Fetch just 5 dependencies
    params = {"limit": 5}
    response = requests.get(f"{BASE_URL}/service_dependencies", headers=headers, params=params)

    if response.status_code != 200:
        print(f"Error: {response.text}")
        return

    data = response.json()
    rels = data.get('relationships', [])

    print(f"Found {len(rels)} relationships.\n")

    if len(rels) > 0:
        print("--- RAW JSON of First Dependency ---")
        print(json.dumps(rels[0], indent=2))
        print("----------------------------------")
    else:
        print("No dependencies found in account? This contradicts earlier results.")

if __name__ == "__main__":
    main()
```

2. Run it

```
python3 debug_json.py
```

3. What to look for

I need to see the output. Specifically, look inside relationships[0].
Does it have source_service? Does it have supporting_service? Does it have dependent_service?

Paste the output here. Once I see the JSON structure, I can give you the corrected Terraform generator script immediately.

```
Found 2 relationships.

--- RAW JSON of First Dependency ---
{
  "id": "D2GM4S41FVBI9NQYPH7",
  "type": "service_dependency",
  "supporting_service": {
    "id": "PJ4Q6F2",
    "type": "technical_service_reference",
    "relationships": null
  },
  "dependent_service": {
    "id": "PA7HPI0",
    "type": "technical_service_reference",
    "relationships": null
  }
}
```

Bot

This confirms the issue! The PagerDuty API fields are named differently than what the Terraform Provider expects, and differently than other endpoints.

- **Your Schema returned:** supporting_service and dependent_service.
- **Terraform expects:** source_service (The Parent) and dependent_service (The Child).

The Valid Mapping:

- **Parent (Upstream):** In the JSON, this is called dependent_service (The service that *has* the dependency). In Terraform, this maps to source_service.
- **Child (Downstream):** In the JSON, this is called supporting_service (The service that *supports* the parent). In Terraform, this maps to dependent_service.

(Naming is confusing here because Terraform uses "dependent_service" to mean "The dependency itself", while the API uses it to mean "The dependent consumer".)

Corrected Python Script (pd_deps_fixed.py)

This script uses the exact fields we found in your debug output.

```
import os
import requests

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_all_items(endpoint, key_name):
    items = []
    params = {'limit': 100, 'offset': 0}
    more = True
    while more:
        r = requests.get(f"{BASE_URL}/{endpoint}", headers=headers, params=params)
        if r.status_code != 200: break
        d = r.json()
        items.extend(d.get(key_name, []))
        more = d.get('more', False)
        params['offset'] += params['limit']
    return items

def main():
    print("1. Mapping Service Names...")
    svc_map = {}

    # Map Technical Services
    for s in get_all_items("services", "services"):
        svc_map[s['id']] = s.get('name') or s.get('summary')

    # Map Business Services
    for s in get_all_items("business_services", "business_services"):
        svc_map[s['id']] = s.get('name') or s.get('summary')

    print(f"   Mapped {len(svc_map)} services.")
    print("2. Fetching Service Dependencies...")
    relationships = get_all_items("service_dependencies", "relationships")

    results = []

    for r in relationships:
        # JSON field: 'dependent_service' is the Consumer/Parent (e.g. Clone Club)
        parent_obj = r.get('dependent_service', {})
        parent_id  = parent_obj.get('id')
        # Check type to determine if it's a business service or technical service
        # API types are usually 'technical_service_reference' or 'business_service_reference'
        parent_api_type = parent_obj.get('type', '')
        parent_tf_type  = "business_service" if "business" in parent_api_type else "service"

        # JSON field: 'supporting_service' is the Provider/Child (e.g. Web App)
        child_obj = r.get('supporting_service', {})
        child_id  = child_obj.get('id')
        child_api_type = child_obj.get('type', '')
        child_tf_type  = "business_service" if "business" in child_api_type else "service"

        results.append({
            "rel_id": r['id'],
            "parent_name": svc_map.get(parent_id, f"Unknown ({parent_id})"),
            "parent_id": parent_id,
            "parent_tf_type": parent_tf_type,
            "child_name": svc_map.get(child_id, f"Unknown ({child_id})"),
            "child_id": child_id,
            "child_tf_type": child_tf_type
        })

    # Output Table
    if not results:
        print("No dependencies found.")
    else:
        print(f"\n{'Parent (Consumer)':<30} | {'Child (Supplier)'}")
        print("-" * 65)
        for res in results:
            print(f"{res['parent_name']:<30} | {res['child_name']}")

    # Generate Terraform
    print("\nGenerating dependencies.tf...")
    with open("dependencies.tf", "w") as f:
        for res in results:
            # Terraform block structure:
            # source_service = The Parent/Consumer
            # dependent_service = The Child/Supplier

            f.write(f"""
resource "pagerduty_service_dependency" "dep_{res['rel_id']}" {{
  dependency {{
    dependent_service {{
      id   = "{res['child_id']}"
      type = "{res['child_tf_type']}"
    }}
    source_service {{
      id   = "{res['parent_id']}"
      type = "{res['parent_tf_type']}"
    }}
  }}
}}
""")
    print("Done. File 'dependencies.tf' created.")

if __name__ == "__main__":
    main()
```

Steps to Finish

1. Run python3 pd_deps_fixed.py.
2. Review the table output in the console to ensure it matches your expectation:
   - *Clone Club* should be the **Parent (Consumer)**.
   - *Clone Club Web App* should be the **Child (Supplier)**.
3. If correct, inspect dependencies.tf.
4. Run the imports for the generated resources using the ID:

   ```bash
   # Example import format
   terraform import pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7 D2GM4S41FVBI9NQYPH7
   ```

Looks good:

```
python3 pd_deps_fixed.py
/Users/rlambie/Library/Python/3.9/lib/python/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
1. Mapping Service Names...
   Mapped 4 services.
2. Fetching Service Dependencies...

Parent (Consumer)              | Child (Supplier)
```

```
--------------------------------------------------------------
Clone Club Web App Service     | OpenShift
Clone Club                     | Clone Club Web App Service

Generating dependencies.tf...
Done. File 'dependencies.tf' created.
rlambie@rlambie-K9XQR37R0J .terraform.d % cat dependencies.tf

resource "pagerduty_service_dependency"
"dep_D2GM4S41FVBI9NQYPH7" {
  dependency {
    dependent_service {
      id   = "PJ4Q6F2"
      type = "service"
    }
    source_service {
      id   = "PA7HPI0"
      type = "service"
    }
  }
}

resource "pagerduty_service_dependency"
"dep_D2GM4TER8JNUZPCH11Z" {
  dependency {
    dependent_service {
      id   = "PA7HPI0"
      type = "service"
    }
    source_service {
      id   = "PJSUJGC"
      type = "business_service"
    }
  }
}
```

Bot

That looks perfect! The output matches your description exactly, and the dependencies.tf file has been generated with the correct IDs and relationship types (business_service vs. service).

You are now ready to finalize this by importing the state so that Terraform explicitly manages these dependencies.

Final Steps to Synchronize

Since the dependencies.tf file is ready, you just need to tell Terraform's state file about these existing resources so it doesn't try to create them again.

Run the following commands in your terminal:

1. Import the first dependency (Web App -> OpenShift)

```
terraform import pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7 D2GM4S41FVBI9NQYPH7
```

2. Import the second dependency (Clone Club -> Web App)

```
terraform import pagerduty_service_dependency.dep_D2GM4TER8JNUZPCH11Z D2GM4TER8JNUZPCH11Z
```

3. Verify
Run a plan to ensure everything is clean. It should show "No changes".

```
terraform plan
```

If the plan comes back clean, you have successfully reverse-engineered your PagerDuty Service Graph into Terraform code! You can now commit dependencies.tf to your version control.

```
terraform import
pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7
D2GM4S41FVBI9NQYPH7
var.pagerduty_token
  Your PagerDuty API Token

  Enter a value:

pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7:
Importing from ID "D2GM4S41FVBI9NQYPH7"...
╷
│ Error: Error importing pagerduty_service_dependency
```

> ### Expecting an importation ID formed as '..'

Bot

It seems the pagerduty_service_dependency resource has a rather quirky requirement for imports. It doesn't accept just the dependency ID (e.g., D2GM4S41FVBI9NQYPH7). Instead, it wants a composite ID string format containing three parts separated by dots.

The required format is:
PXXXXX.service.DYYYYYY

Where:

1. **<supporting_service_id>**: The ID of the Child/Supplier (Use child_id from your generated file).
2. **<supporting_service_type>**: Usually just service or business_service (Use child_tf_type).
3. **<service_dependency_id>**: The actual Relationship ID (Use the dep_... part).

How to Fix the Import Commands

Based on the contents of your dependencies.tf file you shared earlier, here are the exact commands you need to run.

1. Import Web App -> OpenShift

- **Dependency ID:** D2GM4S41FVBI9NQYPH7
- **Child ID (OpenShift):** PJ4Q6F2 (from the dependent_service block)
- **Child Type:** service

Run this command:

```
terraform import pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7 PJ4Q6F2.service.D2GM4S41FVBI9NQYPH7
```

2. Import Clone Club -> Web App

- **Dependency ID:** D2GM4TER8JNUZPCH11Z
- **Child ID (Web App):** PA7HPI0 (from the dependent_service block)

- **Child Type:** service

Run this command:

```
terraform import pagerduty_service_dependency.dep_D2GM4TER8JNUZPCH11Z PA7HPI0.service.D2GM4TER8JNUZPCH11Z
```

(Note: Provide your API token when prompted, or export TF_VAR_pagerduty_token to skip the prompt).

After running these two commands, run terraform plan and it should finally show "No changes".

```
terraform plan
var.pagerduty_token
  Your PagerDuty API Token

  Enter a value:

╷
│ Error: Unsupported block type
│
│   on dependencies.tf line 8, in resource "pagerduty_service_dependency" "dep_D2GM4S41FVBI9NQYPH7":
│    8:     source_service {
│
│ Blocks of type "source_service" are not expected here.
╵

╷
│ Error: Unsupported block type
│
│   on dependencies.tf line 21, in resource "pagerduty_service_dependency" "dep_D2GM4TER8JNUZPCH11Z":
│   21:     source_service {
│
│ Blocks of type "source_service" are not expected here.
╵
```

Bot

You are extremely close! The error is happening because while the PagerDuty **API** uses the term source_service, the **Terraform Provider** uses the term supporting_service.

Also, we need to swap the mapping slightly to match the Terraform definitions:

- **dependent_service**: The service that *has* the dependency (The Parent/Consumer).
- **supporting_service**: The service that *provides* the functionality (The Child/Supplier).

In your generated file, source_service was used (which causes the error), and the IDs were assigned to the blocks based on the API naming, which is slightly different from the Terraform naming.

The Fix

You need to edit your dependencies.tf file. You can do this manually in your text editor since there are only two blocks.

Replace the contents of dependencies.tf with the code below. I have renamed the blocks and ensured the Parent/Child IDs correspond to the correct block type.

```
# 1. Dependency: Web App (Parent) depends on OpenShift (Child)
resource "pagerduty_service_dependency" "dep_D2GM4S41FVBI9NQYPH7" {
  dependency {
    # The Child (Supplier): OpenShift [PJ4Q6F2]
    supporting_service {
      id   = "PJ4Q6F2"
      type = "service"
    }
    # The Parent (Consumer): Web App [PA7HPI0]
    dependent_service {
      id   = "PA7HPI0"
      type = "service"
    }
  }
}

# 2. Dependency: Clone Club (Parent) depends on Web App (Child)
resource "pagerduty_service_dependency" "dep_D2GM4TER8JNUZPCH11Z" {
  dependency {
    # The Child (Supplier): Web App [PA7HPI0]
    supporting_service {
      id   = "PA7HPI0"
      type = "service"
    }
    # The Parent (Consumer): Clone Club [PJSUJGC]
    dependent_service {
      id   = "PJSUJGC"
      type = "business_service"
    }
  }
}
```

Final Verification

1. Save the file.
2. Run terraform plan.

Since you have already successfully imported the state using the IDs in the previous step, Terraform should now look at this config, look at the state, align the two, and report:
"No changes. Your infrastructure matches the configuration."

```
terraform plan
var.pagerduty_token
  Your PagerDuty API Token

  Enter a value:

pagerduty_user.example_user: Refreshing state... [id=P2NMYIE]
pagerduty_service_dependency.dep_D2GM4S41FVBI9NQYPH7: Refreshing state... [id=D2GM4S41FVBI9NQYPH7]
pagerduty_service_dependency.dep_D2GM4TER8JNUZPCH11Z: Refreshing state... [id=D2GM4TER8JNUZPCH11Z]
pagerduty_escalation_policy.cloneclub_policy: Refreshing state... [id=P681258]
pagerduty_service.example_service: Refreshing state... [id=PA7HPI0]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

Bot

That is fantastic news! Seeing **"No changes. Your infrastructure matches the configuration"** is the gold standard in Terraform.

It means you have successfully:

1. Authenticated with the PagerDuty Provider.
2. Defined your core resources (User, Escalation Policy, Service).
3. Reverse-engineered, generated, and imported your existing Service Dependencies graph into code.

You now have a fully manageable Terraform setup for your PagerDuty environment. From this point forward, if you want to add a new dependency or change a timeout, you can just edit the .tf files and run terraform apply.

Nice work debugging those API intricacies

---

don't we need to update the python that generated the dependencies.tf?

Bot

You are absolutely right. Leaving the Python script in a broken state is a recipe for future headaches. If you ever need to regenerate the file or run this against a different account, you'll want the script to produce the correct, valid Terraform code immediately without manual fixing.

Here is the **final, corrected version** of the script.

Changes made:

1. **Block Name Fix:** Changed source_service to dependent_service and dependent_service to supporting_service inside the Terraform block string.
2. **Logic Fix:** ensured the Parent ID (Consumer) goes into dependent_service and the Child ID (Supplier) goes into supporting_service, matching the Terraform Provider's expected schema.

Save this as pd_deps_final_corrected.py:

```
import os
import requests

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}
```

```
def get_all_items(endpoint, key_name):
    items = []
    params = {'limit': 100, 'offset': 0}
    more = True
    while more:
        r = requests.get(f"{BASE_URL}/{endpoint}", headers=headers, params=params)
        if r.status_code != 200: break
        d = r.json()
        items.extend(d.get(key_name, []))
        more = d.get('more', False)
        params['offset'] += params['limit']
    return items

def main():
    print("1. Mapping Service Names...")
    svc_map = {}

    # Map Technical Services
    for s in get_all_items("services", "services"):
        svc_map[s['id']] = s.get('name') or s.get('summary')

    # Map Business Services
    for s in get_all_items("business_services", "business_services"):
        svc_map[s['id']] = s.get('name') or s.get('summary')

    print(f"   Mapped {len(svc_map)} services.")

    print("2. Fetching Service Dependencies...")
    relationships = get_all_items("service_dependencies", "relationships")

    results = []

    for r in relationships:
        # API JSON: 'dependent_service' is the Consumer (Parent)
        parent_obj = r.get('dependent_service', {})
        parent_id  = parent_obj.get('id')
        parent_type_api = parent_obj.get('type', '')
        parent_tf_type  = "business_service" if "business" in parent_type_api else "service"

        # API JSON: 'supporting_service' is the Supplier (Child)
        child_obj = r.get('supporting_service', {})
        child_id  = child_obj.get('id')
        child_type_api = child_obj.get('type', '')
        child_tf_type  = "business_service" if "business" in child_type_api else "service"

        results.append({
            "rel_id": r['id'],
            "parent_name": svc_map.get(parent_id, f"Unknown ({parent_id})"),
            "parent_id": parent_id,
            "parent_tf_type": parent_tf_type,
            "child_name": svc_map.get(child_id, f"Unknown ({child_id})"),
            "child_id": child_id,
            "child_tf_type": child_tf_type
        })

    # Output Table
    if not results:
        print("No dependencies found.")
    else:
        print(f"\n{'Parent (Consumer)':<30} | {'Child (Supplier)'}")
        print("-" * 65)
        for res in results:
            print(f"{res['parent_name']:<30} | {res['child_name']}")

    # Generate Terraform
    print("\nGenerating dependencies.tf...")
    with open("dependencies.tf", "w") as f:
        for res in results:
            # Terraform Provider Mapping:
            # dependent_service block = The entity that HAS the dependency (Parent/Consumer)
            # supporting_service block = The entity satisfying the dependency (Child/Supplier)

            f.write(f"""
resource "pagerduty_service_dependency" "dep_{res['rel_id']}" {{
  dependency {{
    dependent_service {{
      id   = "{res['parent_id']}"
      type = "{res['parent_tf_type']}"
    }}
    supporting_service {{
      id   = "{res['child_id']}"
      type = "{res['child_tf_type']}"
    }}
  }}
}}
""")
    print("Done. File 'dependencies.tf' created.")

if __name__ == "__main__":
    main()
```

---

can we modify this code to identify whether the service is a business or a technical service?

Bot

Yes, absolutely. We can enhance the script to be explicit about the service type in both the console output and the Terraform generation logic.

This version adds a mapped dictionary called type_map so we know exactly what each ID represents (Business vs. Technical) before we even hit the relationship graph. This makes the tf_type assignment much safer than just guessing based on the API string.

Here is the updated script:

```
import os
import requests

# Configuration
API_KEY = os.getenv("PAGERDUTY_TOKEN", "YOUR_API_KEY_HERE")
BASE_URL = "https://api.pagerduty.com"

headers = {
    "Authorization": f"Token token={API_KEY}",
    "Accept": "application/vnd.pagerduty+json;version=2",
    "Content-Type": "application/json"
}

def get_all_items(endpoint, key_name):
    items = []
    params = {'limit': 100, 'offset': 0}
    more = True
    while more:
        r = requests.get(f"{BASE_URL}/{endpoint}", headers=headers, params=params)
        if r.status_code != 200: break
        d = r.json()
        items.extend(d.get(key_name, []))
        more = d.get('more', False)
        params['offset'] += params['limit']
    return items

def main():
    print("1. Mapping Service Names and Types...")
    svc_map = {}

# Stores ID -> Name
    type_map = {}

# Stores ID -> "service" or "business_service"

    # Map Technical Services
    tech_svcs = get_all_items("services", "services")
    for s in tech_svcs:
        svc_map[s['id']] = s.get('name') or s.get('summary')
        type_map[s['id']] = "service"

# Terraform type for technical services

    # Map Business Services
    biz_svcs = get_all_items("business_services", "business_services")
    for s in biz_svcs:
        svc_map[s['id']] = s.get('name') or s.get('summary')
        type_map[s['id']] = "business_service"

# Terraform type for business services

    print(f"   Mapped {len(svc_map)} total services ({len(tech_svcs)} Technical, {len(biz_svcs)}
Business).")

    print("2. Fetching Service Dependencies...")
    relationships = get_all_items("service_dependencies", "relationships")

    results = []

    for r in relationships:
        # API JSON: 'dependent_service' is the Consumer (Parent)
        parent_obj = r.get('dependent_service', {})
        parent_id  = parent_obj.get('id')

        # API JSON: 'supporting_service' is the Supplier (Child)
        child_obj = r.get('supporting_service', {})
        child_id  = child_obj.get('id')

        # Resolve Types using our pre-fetched map (fallback to 'service' if unknown)
        parent_tf_type = type_map.get(parent_id, "service")
        child_tf_type = type_map.get(child_id, "service")

        results.append({
            "rel_id": r['id'],
            "parent_name": svc_map.get(parent_id, f"Unknown ({parent_id})"),
            "parent_id": parent_id,
            "parent_tf_type": parent_tf_type,
            "child_name": svc_map.get(child_id, f"Unknown ({child_id})"),
            "child_id": child_id,
            "child_tf_type": child_tf_type
        })

    # Output Table with Types
    if not results:
        print("No dependencies found.")
    else:
        # Format string for nice columns
        row_fmt = "{:<30} {:<15} | {:<30} {:<15}"
        print("\n" + row_fmt.format("Parent (Consumer)", "[Type]", "Child (Supplier)", "[Type]"))
        print("-" * 95)
        for res in results:
            # Shorten types for display
            p_type_display = "Business" if res['parent_tf_type'] == "business_service" else
"Technical"
            c_type_display = "Business" if res['child_tf_type'] == "business_service" else
"Technical"

            print(row_fmt.format(
                res['parent_name'][:29],
                p_type_display,
                res['child_name'][:29],
                c_type_display
            ))

    # Generate Terraform
    print("\nGenerating dependencies.tf...")
    with open("dependencies.tf", "w") as f:
        for res in results:
            f.write(f"""
resource "pagerduty_service_dependency" "dep_{res['rel_id']}" {{
  dependency {{
    dependent_service {{
      id   = "{res['parent_id']}"
      type = "{res['parent_tf_type']}"
    }}
    supporting_service {{
      id   = "{res['child_id']}"
      type = "{res['child_tf_type']}"
    }}
  }}
}}
""")
    print("Done. File 'dependencies.tf' created/updated.")
    # Created/Modified files during execution:
    print("dependencies.tf")

if __name__ == "__main__":
    main()
```