**Terraform Scenarios-**

**Every DevOps Engineer Should Know**

Terraform is quite powerful tool but even the best setups can run across problems. I have compiled a list of 5 real-world scenarios that every DevOps engineer should be ready & prepared for:

1. State file deleted
2. Two engineers run `terraform apply` at the same time
3. Apply fails halfway through
4. AWS API rate limits hit during deployment
5. Manual infrastructure changes cause drift



These are common challenges encountered in real projects. If not handled correctly, they can lead to service outages, failed deployments, and hours of unnecessary debugging.

**1. State file deleted**

If the **Terraform state file is deleted**, it can cause **serious issues**, because Terraform relies heavily on the state file (terraform.tfstate) to understand the **current state of your infrastructure**.

➣ ***Once State File is Deleted:***

**A. Terraform Loses Track of Resources**:

Terraform no longer knows what resources have already been created.

It will assume **nothing exists**, and the next terraform apply will try to **create everything again**.

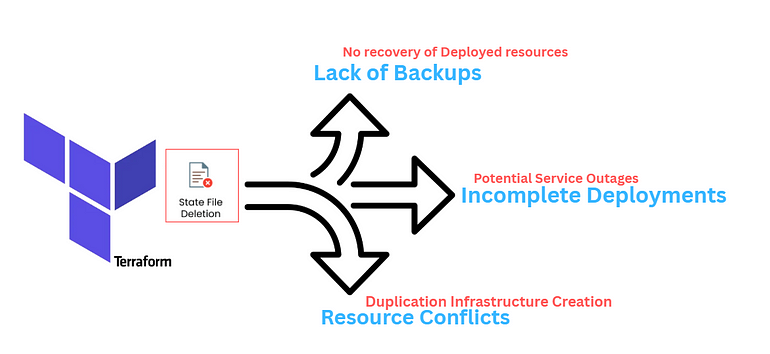
**B. Risk of Duplicate Resources**:

Since Terraform has no memory of the existing infrastructure, it may **recreate resources** (e.g., EC2 instances, S3 buckets, etc.).

This can result in **duplicate resources**, **conflicts**, or even **downtime** if the recreated resources interfere with the existing ones.

**C. Orphaned Resources**:

Existing resources are now considered **orphaned**, and Terraform can’t manage or destroy them anymore unless they are re-imported.



**2. Two engineers run `terraform apply` at the same time**

If **two engineers run terraform apply at the same time** on the same Terraform state, it can cause conflicts and potentially**corrupt the state file**.

➣One of any event does occure:

**A. Race Condition**:

Both engineers read the **current state file** at the same time.Both try to make changes based on that **same initial state**.

When they try to write back the new state, one will **overwrite** the other’s changes.

**B. Terraform Locking Mechanism**:

By default, **Terraform uses a state lock** mechanism to prevent concurrent operations.

This works **only if you’re using a remote backend that supports locking**, like:

* AWS S3 with DynamoDB for locking
* Terraform Cloud or Enterprise
* Google Cloud Storage with locking

If you’re using **local state**, there’s **no locking**, and you’ll likely run into **state corruption**.

**C. State Corruption Risk**:

If two processes write to the state simultaneously, the file can get corrupted.

Manual intervention would be needed to restore from backup or fix the issue.

**3. Apply fails halfway through**

If a **terraform apply**fails halfway through, it can leave your infrastructure in a partially applied and inconsistent state.



➣ **When terraform apply Fails Midway:**

**A. Some Resources Are Created/Modified, Others Aren’t**:

Terraform executes changes **sequentially** (or in parallel where possible).

If it fails during the apply process, some resources may already be **provisioned, modified, or destroyed**, while others are not.

**B. Terraform State File Might Be Updated**:

Terraform updates the state **only after** each successful resource change.

So if the apply fails after a few steps, the state file reflects only the successful changes.

This causes a **mismatch between your .tf code and actual infra**.

**4. AWS API rate limits hit during deployment**

If your **Terraform deployment hits AWS API rate limits**, the apply process can **fail**, **hang**, or **retry excessively**, depending on how Terraform and AWS handle the situation.

**➣ AWS API Rate Limits means**

AWS has **throttling limits** for each service’s API calls (like EC2, IAM, S3, etc.).

If you exceed the allowed number of requests per second, AWS returns a:

***ThrottlingException: Rate exceeded***

Terraform might show:

***Error: Error launching source instance: RequestLimitExceeded***

**➣ Why Does This Happen with Terraform?**

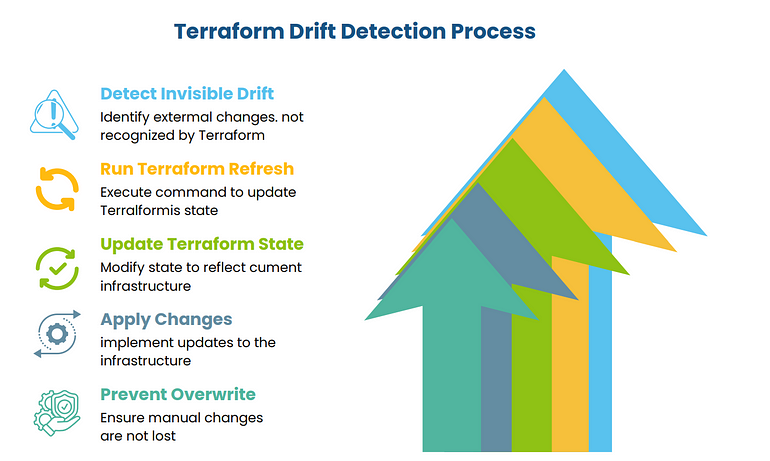
Terraform may:

* Create many resources in **parallel** (by default)
* Repeatedly call APIs (e.g., tagging, checking status)
* Trigger multiple retries if backend modules are reused or looped with count/for\_each

**5. Manual infrastructure changes cause drift**

When someone makes **manual changes to the infrastructure outside Terraform** (via the AWS console, CLI, etc.), it causes something called **“drift”** — and that can lead to **unpredictable behavior**.

**Drift** = when the **real infrastructure state** is different from what’s defined in your .tf code and recorded in the terraform.tfstate file.



**How Does Drift Happen?**

* You **manually change an EC2 instance type** from t2.micro to t3.small.
* Someone **modifies a security group** via the AWS console.
* A tag is removed from a resource.

Terraform still thinks things are as defined in your .tf files and tfstate, but the **reality is different**.

*Invisible drift detected! Terraform won’t notice external changes unless you run terraform refresh. Future applies might overwrite manual updates without warning. Always detect drift before applying!*