Apex Trigger Best Practices: From Anti-Patterns to a Maintainable Framework

CHEATSHEET

Transform monolithic, unmaintainable triggers into a bulk-safe, single-trigger framework that scales.

Seventeen triggers on Account, zero shared handler logic, and a LimitException on Friday afternoon? It's time to build a maintainable trigger framework.

1 The Cost of Apex Technical Debt

Unstructured triggers compound quickly into unmaintainable legacy code. Watch out for these four common anti-patterns during code reviews:

- SOQL and DML operations inside loops (causes LimitException: 101)
- Business logic written directly inside the trigger file
- Multiple triggers on the same SObject causing unpredictable execution order
- Treating only Trigger.new[0] instead of fully bulkifying logic

2 The Fatal Anti-Pattern: SOQL in Loops

This code executes perfectly with 1 record but instantly fails at 200 records with a 'Too many SOQL queries: 101' governor limit exception.

APEX

```
trigger AccountTrigger on Account (before insert) {
    for (Account acc : Trigger.new) {
        // SOQL in loop - fires for every record in tran
saction
        List contacts = [SELECT Id FROM Contact WHERE Ac
countId = :acc.Id];
    }
}
```

3 The One-Trigger-Per-Object Pattern

The absolute baseline of standard Apex architecture is enforcing a single trigger per object. This serves as your routing engine, not your execution layer.

- Trigger file contains zero business logic
- Only delegates events to a dedicated Handler class
- Ensures deterministic, predictable execution order
- Simplifies deployment and isolated unit testing

4 Key Components of a Robust Framework

A production-grade trigger framework requires three essential layers to ensure safety, performance, and maintainability:

- Bulk-safe design that handles collection processing seamlessly
- Handler classes for isolation of business logic
- Recursive guards to prevent endless execution loops
- Governor limit hygiene using Maps and Sets for bulkification