The Apex Governor-Limit Survival Kit

15 battle-tested patterns that keep Apex alive when production hands it 5,000 records instead of the 5 you tested with. Copy, adapt, deploy.

Triggers & bulkification | SOQL / DML limits | Async & Batch | Callouts | Testing at scale

WHAT'S INSIDE — ALL 15 PATTERNS

Compiled by the Genetrix engineering team · genetrix.tech

“It worked in the sandbox” is the most expensive sentence in Salesforce development. It worked because you tested with five records. Every pattern below is the fix for a specific governor limit that only shows up at volume — the day a data load, integration, or Flow pushes real numbers through your code.

01 Bulkify every trigger — design for 200, not 1

SOQL · DML

THE TRAP Logic written for a single record dies the moment a data load, API call, or Flow updates records in bulk.

```
// One-line trigger -> all logic in a bulk-safe handler
trigger ContactTrigger on Contact (before insert, before update) {
    new ContactTriggerHandler().run();        // never loop-and-DML in the trigger
}

public with sharing class ContactTriggerHandler {
    public void run() {
        // Trigger.new is ALWAYS a collection - operate on the whole set
        for (Contact c : (List<Contact>) Trigger.new) {
            if (c.LastName != null) c.LastName = c.LastName.trim(); // in-memory only
        }
    }
}
```

WHY IT WORKS One thin trigger → a handler that only ever touches collections. No SOQL/DML in the trigger body, ever.

02 Never put SOQL inside a loop

TOO MANY SOQL QUERIES: 101

THE TRAP A query inside a for-loop runs once per record. 101 records = a hard failure for every user.

```
// BAD: 101 error waiting to happen
for (Opportunity o : Trigger.new) {
    Account a = [SELECT Industry FROM Account WHERE Id = :o.AccountId]; // SOQL in loop
}

// GOOD: query once, look up from a Map
Set<Id> acctIds = new Set<Id>();
for (Opportunity o : Trigger.new) acctIds.add(o.AccountId);

Map<Id, Account> acctById = new Map<Id, Account>(
    [SELECT Id, Industry FROM Account WHERE Id IN :acctIds]
);
for (Opportunity o : Trigger.new) {
    Account a = acctById.get(o.AccountId);
    if (a != null) o.Description = a.Industry;
}
```

WHY IT WORKS Query once with a Set of Ids, drop the rows into a Map, and look up in O(1) inside the loop.

03 Never put DML inside a loop

TOO MANY DML STATEMENTS: 151

THE TRAP update inside a loop spends one of your 150 DML statements per iteration.

```
// BAD: one update() per iteration
for (Case c : scope) { c.Status = 'Closed'; update c; }

// GOOD: collect, then a single DML
List<Case> toUpdate = new List<Case>();
for (Case c : scope) { c.Status = 'Closed'; toUpdate.add(c); }
if (!toUpdate.isEmpty()) update toUpdate;
```

WHY IT WORKS Collect rows in a List, fire a single DML after the loop. One statement, thousands of rows.

04 Write selective, least-privilege SOQL

QUERY ROWS: 50,000 · FLS

THE TRAP SELECT every field, no filter, no limit — slow, heap-hungry, and a non-selective query error at scale.

```
// BAD: unbounded, every field, no security
List<Account> a = [SELECT FIELDS(ALL) FROM Account];

// GOOD: needed fields, indexed filter, bounded, user-mode security
List<Account> a = [
    SELECT Id, Name, Industry
    FROM Account
    WHERE CreatedDate = LAST_N_DAYS:30     // indexed + selective
    WITH USER_MODE                          // enforces FLS + sharing
    ORDER BY CreatedDate DESC
    LIMIT 200
];
```

WHY IT WORKS Only the fields you use, an indexed WHERE, a LIMIT, and WITH USER_MODE to enforce FLS + sharing for free.

05 Group children once with Map<Id, List<>>

CPU · SOQL

THE TRAP Re-querying a parent's children every time you need them is the slow road to a CPU timeout.

```
Map<Id, List<Contact>> byAccount = new Map<Id, List<Contact>>();
for (Contact c : [SELECT Id, AccountId FROM Contact WHERE AccountId IN :acctIds]) {
    if (!byAccount.containsKey(c.AccountId))
        byAccount.put(c.AccountId, new List<Contact>());
    byAccount.get(c.AccountId).add(c);
}
// O(1) lookups instead of nested queries
List<Contact> kids = byAccount.get(someAccountId);
```

WHY IT WORKS Build the parent→children map a single time and reuse it everywhere downstream.

06 Move heavy work + callouts to Queueable

ASYNC · CALLOUTS

THE TRAP Doing slow work or external callouts inside the user's transaction blows CPU and blocks the UI.

```apex
public class LeadEnrichmentJob implements Queueable, Database.AllowsCallouts {
    private List<Id> leadIds;
    public LeadEnrichmentJob(List<Id> leadIds) { this.leadIds = leadIds; }

    public void execute(QueueableContext ctx) {
        // heavy work + external callouts, off the user transaction
        // ... process this.leadIds ...
        if (!Test.isRunningTest() && moreToProcess)
            System.enqueueJob(new LeadEnrichmentJob(nextBatch));   // chain
    }
}
// Kick off: System.enqueueJob(new LeadEnrichmentJob(ids));
```

WHY IT WORKS Queueable runs async with its own limits, takes sObjects, and chains to itself for large jobs.

07 Process millions of rows with Batchable + Stateful

50M+ ROWS

THE TRAP A normal transaction can touch 10,000 rows. Past that you need batches that each get fresh limits.

```apex
public class DormantContactBatch
        implements Database.Batchable<SObject>, Database.Stateful {
    public Integer processed = 0;            // survives across batches

    public Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator(
            'SELECT Id FROM Contact WHERE LastActivityDate < LAST_N_DAYS:365');
    }
    public void execute(Database.BatchableContext bc, List<Contact> scope) {
        for (Contact c : scope) c.Status__c = 'Dormant';
        update scope;                        // each scope = its own limits
        processed += scope.size();
    }
    public void finish(Database.BatchableContext bc) { System.debug(processed); }
}
// Database.executeBatch(new DormantContactBatch(), 200);
```

WHY IT WORKS Each execute() scope is its own governor context; Database.Stateful carries counters across batches.

08 Prefer Queueable over @future

ASYNC DESIGN

THE TRAP @future is fire-and-forget: primitives only, no chaining, and almost impossible to monitor.

```
// @future: primitives only, NO chaining, hard to track
@future(callout=true)
public static void syncIds(Set<Id> ids) { /* ... */ }

// Prefer Queueable: sObjects, chainable, gives you a job Id to monitor
System.enqueueJob(new SyncJob(ids));
// Rule of thumb: new async work -> Queueable.
```

WHY IT WORKS Queueable accepts sObjects, returns a trackable job Id, and chains. Keep @future for legacy only.

09 Stop trigger recursion with a static guard

CPU · SOQL · DML

THE TRAP A trigger that updates its own object re-fires itself, multiplying every query until you hit a limit.

```
public class TriggerGuard {
    private static Set<Id> done = new Set<Id>();
    // returns only the Ids not yet processed in this transaction
    public static List<Id> newOnes(Set<Id> ids) {
        List<Id> fresh = new List<Id>();
        for (Id i : ids) if (!done.contains(i)) { fresh.add(i); done.add(i); }
        return fresh;
    }
}

// In the handler:
List<Id> fresh = TriggerGuard.newOnes(Trigger.newMap.keySet());
if (fresh.isEmpty()) return;
```

WHY IT WORKS A static Set remembers which Ids were processed this transaction and skips the re-entry.

10 Guard limits + ship a kill switch

DEFENSIVE · CALLOUTS

THE TRAP When a batch nears a limit it fails hard — and you can't disable a misbehaving trigger without a deploy.

```
// Defer instead of failing when you are near the ceiling
if (Limits.getCallouts() >= Limits.getLimitCallouts()) {
    System.enqueueJob(new DeferredJob(records));
    return;
}

// Org/profile/user kill switch - toggle without deploying
if (AutomationSettings__c.getInstance().Disable_Triggers__c) return;
```

WHY IT WORKS Check Limits before risky work and hand off; gate automation behind a hierarchy Custom Setting.

11 Aggregate in SOQL, not in Apex

HEAP · QUERY ROWS

THE TRAP Pulling thousands of rows just to sum them in a loop burns heap and query rows for nothing.

```
// BAD: rows pulled into Apex just to total them
Integer total = 0;
for (Opportunity o : [SELECT Amount FROM Opportunity WHERE AccountId = :id])
    total += (o.Amount == null ? 0 : o.Amount.intValue());

// GOOD: let the database aggregate
AggregateResult ar = [
    SELECT SUM(Amount) total FROM Opportunity WHERE AccountId = :id
];
Decimal total = (Decimal) ar.get('total');
```

WHY IT WORKS SUM/COUNT/GROUP BY run in the database and return one row instead of thousands.

12 Cache static reference data in Platform Cache

SOQL · CPU

THE TRAP Querying the same config / mapping table every transaction is pure governor-limit waste.

```
Cache.OrgPartition part = Cache.Org.getPartition('local.config');
Map<String, String> settings = (Map<String, String>) part.get('appSettings');

if (settings == null) {
    settings = loadSettingsFromDB();            // query once
    part.put('appSettings', settings, 3600);    // cache for 1 hour
}
// use settings ...
```

WHY IT WORKS Read it once, store it in the Org partition with a TTL, and serve it from memory after that.

13 Use partial-success DML so one bad row can't roll back 9,999

DML · UX

THE TRAP A single invalid record in a bulk update throws and rolls back the entire batch.

```
Database.SaveResult[] results = Database.update(records, false); // allOrNone = false
for (Integer i = 0; i < results.size(); i++) {
    if (!results[i].isSuccess()) {
        for (Database.Error e : results[i].getErrors()) {
            System.debug(records[i].Id + ': ' + e.getStatusCode() + ' - ' + e.getMessage());
        }
    }
}
```

WHY IT WORKS allOrNone = false commits the good rows and hands you a SaveResult for the failures to log/retry.

14 Make callouts bulk-safe — and mock them in tests

TOO MANY CALLOUTS: 101

THE TRAP A callout per record hits the limit fast; and tests fail outright if a callout isn't mocked.

```
@isTest
private class SyncServiceTest {
    @isTest static void pushesInOneCallout() {
        Test.setMock(HttpCalloutMock.class, new MockResponse(200, '{"ok":true}'));
        Test.startTest();
        SyncService.pushAll(ids);          // ONE bulk callout inside
        Test.stopTest();
        System.assertEquals(1, SyncService.calloutCount);
    }
}
```

WHY IT WORKS Aggregate into one callout, and set an HttpCalloutMock so the test never leaves the org.

15 Prove bulk-safety with a 200-record test

COVERAGE · CONFIDENCE

THE TRAP Tests that insert one record pass happily while the code silently breaks at volume in production.

```
@isTest
public class TestDataFactory {
    public static List<Account> accounts(Integer n) {
        List<Account> a = new List<Account>();
        for (Integer i = 0; i < n; i++) a.add(new Account(Name = 'Acme ' + i));
        insert a;
        return a;
    }
}

@isTest static void survivesBulk() {
    List<Account> accts = TestDataFactory.accounts(200);   // not 1
    Test.startTest();
    update accts;
    Test.stopTest();
    System.assertEquals(200, [SELECT COUNT() FROM Account WHERE Name LIKE 'Acme%']);
}
```

WHY IT WORKS A test data factory + a 200-record assertion is the only way to prove your code is actually bulkified.

Inherited an org that hits limit errors at volume?

That's most of what we clean up. Genetrix re-architects Apex, triggers, and integrations to run at enterprise scale across Salesforce core, Marketing Cloud, Data Cloud, and Agentforce.

Book a 30-minute code review →