Multi-Squad Git Flow Strategy

Enterprise API Gateway Architecture with AWS Lambda

Document Type:	Technical Architecture Guide
Version:	1.0
Date:	January 09, 2026
Target Audience:	Development Teams, DevOps, Platform Engineers
Technology Stack:	AWS API Gateway, Lambda, Terraform, GitHub Actions

Table of Contents

1. Executive Summary

This document outlines an enterprise-grade Git Flow strategy adapted for multi-squad development environments working with AWS API Gateway and Lambda functions. The strategy eliminates common pain points such as merge conflicts, cherry-picking nightmares, and feature integration issues while maintaining compliance with existing Git Flow branching models.

- ✓ Zero cherry-picking required
- ✓ Independent squad development
- ✓ Isolated testing environments per squad
- ✓ Clean Git history maintained
- ✓ Compliant with existing Git Flow structure
- ✓ Production-ready from day one

2. The Challenge: Multi-Squad Development

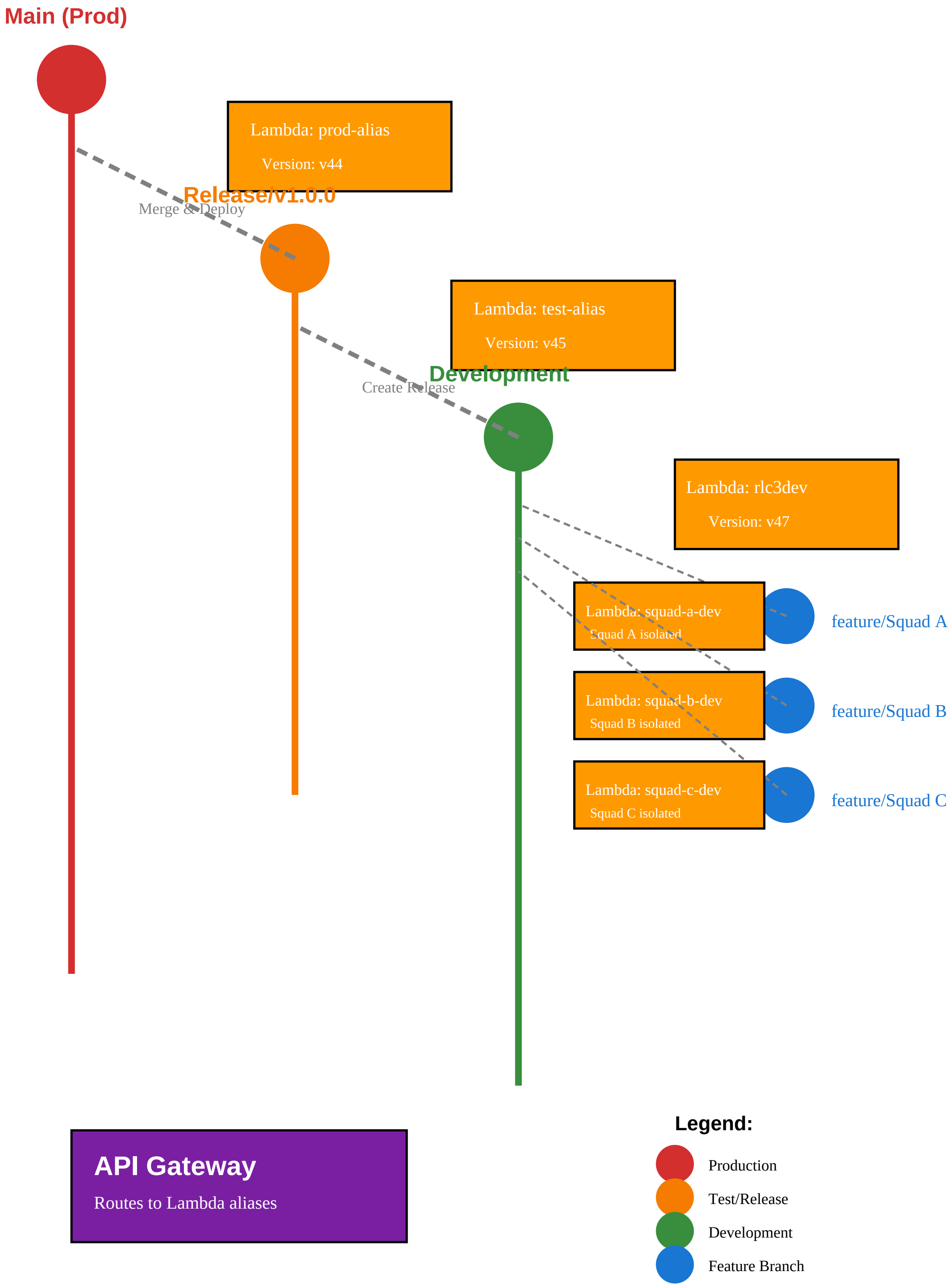
Organizations with multiple development squads working on the same API Gateway infrastructure face several critical challenges:

- Multiple teams merging features to a shared Development branch simultaneously
- Selective release requirements where not all features are ready for production
- Cherry-picking creating merge conflicts and tracking nightmares
- Feature flags adding code complexity and technical debt
- Testing isolation issues when multiple features are in progress

Traditional solutions fail because: They either require extensive code refactoring (feature flags), create git history chaos (cherry-picking), or block independent squad progress (release trains).

3. Your Current Git Flow Structure

Multi-Squad Git Flow Strategy



Your existing Git Flow follows the industry-standard model with five branch types:

<b>Branch Type</b>	<b>Purpose</b>	<b>Deployment Target</b>
Main (Prod)	Production-ready code	Production AWS Account
Release/vX.X.X	Release candidate testing	Test AWS Account
Development	Integration environment	Dev AWS Account (shared)
feature/*	New feature development	Squad-specific dev stages
bugfix/*	Bug fixes from release	Test AWS Account

4. The Problem Scenarios

4.1 Scenario: Selective Release Chaos

Monday:
- Squad A merges `feature/checkout` → Development
- Squad B merges `feature/payment` → Development
- Squad C merges `feature/refund` → Development

Tuesday: Create `release/v1.0.0` from Development
- ALL 3 features are now in release branch
- Squad B's payment feature has a critical bug discovered in testing!

The Dilemma:

■ Cherry-pick Squad A + C	Creates merge conflicts, breaks traceability
■ Revert Squad B in Development	Breaks Development environment for Squad B
■ Delay entire release	Squad A and C blocked waiting for Squad B
■ Use feature flags	Code complexity, technical debt, cleanup burden

5. Enterprise Solution: Release Readiness Gates

The solution is to control **WHAT** merges to Development, not manipulate code after merge. Development branch becomes a **pre-release staging area**, not a playground.

Core Principle:

Features only merge to Development when they are release-ready.

This means: passing tests, security scans approved, product owner sign-off, and ready for production deployment.

Key Benefits:

✓ Squad B keeps working in `feature/squad-b-payment` (not merged)

✓ Squad A and C merge to Development (release-ready)

✓ Release branch created from Development contains only A + C

✓ **Zero cherry-picking** - Squad B simply wasn't merged yet

✓ Squad B continues working independently in their feature branch

6. Implementation Strategy

6.1 Branch Protection Rules (GitHub)

Configure Development branch with strict protection rules to enforce release readiness:

Rule	Configuration	Purpose
Required Reviews	2 approvers minimum	Code quality validation
Status Checks	pytest, security-scan, contract-tests	Automated quality gates
Manual Gate	product-owner-approval	Business readiness confirmation
JIRA Integration	Status = 'Ready for Release'	Workflow state validation
Branch Currency	Must be up-to-date with Development	Prevent integration issues

6.2 Long-Lived Feature Branches

Unlike traditional trunk-based development, Git Flow allows (and encourages) feature branches to live for extended periods until ready:

`feature/squad-b-payment` (NOT merged to Development)
↓
Rebases from Development daily (stays current with Squad A + C changes)
↓
Deploys to `squad-b-dev` stage (isolated testing)
↓
When ready: PR to Development → All gates pass → Merge

Critical Practice: Daily Rebasing

```
$ git checkout feature/squad-b-payment
$ git fetch origin development
$ git rebase origin/development # Stay current with other squads
$ git push origin feature/squad-b-payment --force-with-lease
```

6.3 Per-Squad Development Environments (AWS)

Each squad gets an isolated API Gateway stage + Lambda alias in the Dev AWS Account. This eliminates testing conflicts and allows true parallel development.

Terraform Configuration:

```
# Lambda Function (single codebase, multiple aliases)
resource "aws_lambda_function" "motor_quote" {
function_name = "motor-quote-api"
runtime = "python3.12"
handler = "app.lambda_handler"
}

# Squad A Development Alias
resource "aws_lambda_alias" "squad_a_dev" {
name = "squad-a-dev"
function_name = aws_lambda_function.motor_quote.function_name
function_version = "$LATEST"
}

# Squad B Development Alias
resource "aws_lambda_alias" "squad_b_dev" {
name = "squad-b-dev"
function_name = aws_lambda_function.motor_quote.function_name
function_version = "$LATEST"
}

# API Gateway Stage for Squad A
resource "aws_api_gateway_stage" "squad_a_dev" {
stage_name = "squad-a-dev"
rest_api_id = aws_api_gateway_rest_api.motor.id
deployment_id = aws_api_gateway_deployment.motor.id

variables = {
lambda_alias = "squad-a-dev"
}
}
```

Environment URLs:

Squad	API Endpoint	Lambda Alias
Squad A	https://api.dev.company.com/squad-a-dev/v1/quote	squad-a-dev
Squad B	https://api.dev.company.com/squad-b-dev/v1/quote	squad-b-dev
Squad C	https://api.dev.company.com/squad-c-dev/v1/quote	squad-c-dev
Shared Dev	https://api.dev.company.com/rlc3dev/v1/quote	rlc3dev

7. Complete Workflow Example

Week 1-2: Feature Development

- Squad A: Works in `feature/squad-a-checkout` → Deploys to squad-a-dev
- Squad B: Works in `feature/squad-b-payment` → Deploys to squad-b-dev
- Squad C: Works in `feature/squad-c-refund` → Deploys to squad-c-dev
- **Result:** Zero conflicts, parallel development, isolated testing

Week 2 End: Merge to Development

- ■ Squad A: Feature ready → PR to Development → **Merged**
- ■ Squad C: Feature ready → PR to Development → **Merged**
- ■■ Squad B: Not ready → **Stays in feature branch**
- **Development branch = Squad A + Squad C only**

Week 3: Release to Test

- Create `release/v1.0.0` from Development
- Deploy to Test AWS Account → Tag: v1.0.0rc1
- QA validates Squad A + C features
- Squad B continues development in parallel (not blocked)

Week 4: Production Deployment

- Merge `release/v1.0.0` to Main
- Deploy to Production → Tag: v1.0.0
- Squad A + C features go live
- Squad B targets next release (v1.1.0)

8. AWS Architecture Integration

The Git Flow strategy integrates seamlessly with AWS Lambda versioning and API Gateway stages:

Git Branch	Deployment Target	Lambda Alias	API Gateway Stage
feature/squad-a-*	Dev Account	squad-a-dev → v47	squad-a-dev
feature/squad-b-*	Dev Account	squad-b-dev → v48	squad-b-dev
development	Dev Account	rlc3dev → v49	rlc3dev
release/v1.0.0	Test Account	test → v45	test
main (prod)	Prod Account	prod → v44	prod

Lambda Version Management:

Each Git branch push triggers Lambda version publishing and alias updates:

```
# GitHub Actions workflow
- name: Publish Lambda Version
run: |
VERSION=$(aws lambda publish-version \
--function-name motor-quote-api \
--query 'Version' --output text)

aws lambda update-alias \
--function-name motor-quote-api \
--name ${{ env.STAGE_ALIAS }} \
--function-version $VERSION
```

9. Key Changes Required

To implement this strategy in your organization, the following changes are necessary:

Infrastructure Changes

✓ Create per-squad Lambda aliases in Dev AWS Account

✓ Create per-squad API Gateway stages

✓ Configure stage variables to route to correct aliases

✓ Set up S3 buckets for Lambda package storage per environment

GitHub Configuration

✓ Enable branch protection on Development with required status checks

✓ Configure JIRA integration for 'Ready for Release' workflow state

✓ Set up GitHub Actions workflows for branch-based deployments

✓ Create deployment approvals for Test and Prod environments

Team Process Changes

✓ Squads maintain feature branches until release-ready

✓ Daily rebase practice from Development branch

✓ Product Owner approval required before Development merge

✓ Release manager creates release branches from Development

CI/CD Pipeline Updates

✓ Branch name detection logic for routing deployments

✓ Automated Lambda version publishing and alias updates

✓ Integration tests per squad environment

✓ Rollback automation with CloudWatch alarm integration

10. Conclusion

This multi-squad Git Flow strategy provides a practical, enterprise-grade solution that:

- Maintains your existing Git Flow structure (no radical changes)
- Eliminates cherry-picking and merge conflicts entirely
- Enables true parallel squad development with zero interference
- Leverages AWS Lambda aliases for environment isolation
- Provides clear release gates through branch protection
- Scales to dozens of squads without process breakdown
- Maintains full audit trail and compliance requirements
- Enables instant rollback through Lambda version management

Bottom Line: By controlling what merges to Development (release-ready features only) and providing isolated development environments per squad, you achieve conflict-free multi-squad development without code complexity or git manipulation.

Next Steps

1. Review this strategy with platform engineering and squad leads
2. Provision per-squad AWS infrastructure (Lambda aliases, API Gateway stages)
3. Configure GitHub branch protection rules on Development branch
4. Update CI/CD pipelines for branch-based routing
5. Pilot with 2-3 squads before full rollout
6. Document runbooks for common scenarios (bugfixes, hotfixes, rollbacks)