The Hidden Dangers of Non-Deterministic Feature Flags

Abhishek Tiwari

Citation: *A. Tiwari*, "The Hidden Dangers of Non-Deterministic Feature Flags", Abhishek Tiwari, 2026. doi:10.59350/xnkw5-9pm76

Published on: May 01, 2026

Feature flags have become the backbone of modern software deployment strategies. They enable teams to deploy code continuously, run experiments, and gradually roll out new features safely to users. But when feature flags incorporate non-deterministic behaviour, particularly percentage-based rollouts, they can introduce subtle bugs that are maddeningly difficult to detect and debug.

Feature flags or toggles are meant to reduce risk and increase safety during deployments ([1]), yet when implemented with non-deterministic behaviour in distributed systems, they can actually **amplify risk** and create emergent failure modes that wouldn't exist without them.

After years of battle scars from production incidents, I've learned that non-deterministic feature flags are powerful tools that demand respect. This post explores the mathematical foundations, common anti-patterns, emergent behaviours, and real-world bugs that arise when these flags go wrong.

What Makes a Feature Flag Non-Deterministic?

A deterministic feature flag produces the same output for the same input every time. Turn it on for user ID 12345, and user 12345 will always see that feature, no matter which server handles the request, no matter what time it is, no matter how many times you ask. This predictability is what makes feature flags feel safe.

Non-deterministic flags break this contract. They introduce variability through mechanisms like **percentage-based rollouts** ("enable for 20% of users"), **time-based activation** ("only during business hours"), **randomized group selection**, **context-dependent logic** that responds to server load or geography, and **distributed evaluation** where different services independently assess the same conceptual flag.

Each of these mechanisms offers genuine flexibility. Percentage rollouts let you validate a feature against a small user cohort before committing. Time-based flags let you align feature exposure with support staffing. But each mechanism also introduces a surface area for inconsistency, and these surfaces tend to compound dangerously in production.

The Math and the Mayhem

Hash-Based Distribution

Most percentage-based feature flags rely on hash functions to create a deterministic-but-seemingly-random distribution. The core idea is straightforward: given a user identifier and a feature name, produce a hash value, then check whether it falls within the target percentage window.

For a rollout targeting p% of users, a user is included when:

$$h(\text{userId} \parallel \text{featureName}) \bmod 100 < p$$

where $h(\cdot)$ is a hash function and $\parallel$ denotes string concatenation. The hash function serves two purposes simultaneously: it creates a *stable* assignment (the same user always lands in the same bucket) and an *apparently uniform* distribution across users.

Crucially, the feature name is included in the hash input so that independent flags produce statistically independent assignments. E.g., a user who falls in the 20th percentile for the `new_ui` flag is not guaranteed to fall in the 20th percentile for the `new_backend` flag.

```
function isFeatureEnabled(userId, featureName, percentage) {
  const hash = djb2Hash(userId + featureName); // feature name salts the
     hash
  return (hash % 100) < percentage;
}


// A user's assignment is stable but independent across flags
isFeatureEnabled('user_42', 'new_ui',      20); // → true
isFeatureEnabled('user_42', 'new_backend', 20); // → false  (different
    salt = different bucket)
```

Where Uniformity Breaks Down

This design is elegant in theory, but real implementations must grapple with several failure modes. Most hash functions are designed for speed, not statistical perfection. The djb2 algorithm and similar lightweight hash functions can produce non-uniform distributions when applied to structured inputs like short strings. When you compute $h(\text{key}) \bmod 100$, any clustering in the hash space translates directly into clustering in your user assignment.

The practical consequence is that your “20% rollout” might actually reach 17% or 24% of users, depending on the distribution of your user IDs and the quality of your hash function. For small user bases, this deviation can be substantial.

Safety Mechanisms That Create Danger

Feature flags were designed to make deployments safer along several dimensions: gradual rollouts let you catch bugs before they affect everyone; quick rollback avoids full redeployments; A/B testing validates changes before full release; and isolating risk to small populations limits blast radius.

But non-deterministic feature flags can invert this safety promise, particularly in distributed systems. The problem is combinatorial explosion. A single feature flag creates two code paths. Two flags create

four. N flags create 2^N paths. This is manageable at small N, but most teams run far more flags than they realise, and almost no team tests the full interaction space.

If you have ten simultaneous flags, you have $2^{10} = 1024$ possible system states. Testing all of them is impractical. Most teams test the "all on" and "all off" states and perhaps a handful of targeted combinations, leaving the vast majority of production states untested.

The very mechanism meant to reduce risk becomes a source of systemic risk.

Emergent Non-Determinism: When Flags Interact

The most dangerous bugs arise not from individual flags but from their **emergent interactions**. This is where non-determinism compounds in unexpected ways.

The Cascading Percentage Problem

Consider three independent 50% rollouts running simultaneously: a new UI, a new backend, and a new cache layer. Each flag is evaluated independently, so a user's assignment to one flag has no bearing on their assignment to another. The resulting distribution across all combinations is:

Combination	Probability
old-old-old	$0.5^3 = 12.5\%$
new-old-old	12.5%
old-new-old	12.5%
old-old-new	12.5%
new-new-old	12.5%
new-old-new	12.5%
old-new-new	12.5%
new-new-new	12.5%

The math is simple, but the operational reality is alarming. You now have eight distinct system configurations running simultaneously in production. Have you tested what happens when the new UI communicates with the old backend through the new cache? What about the new UI with the new backend and the old cache? Most teams test two of these eight combinations. The other six are live experiments on your production users.

Implicit Dependencies Create Emergent Failures

The cascading problem becomes catastrophic when flags have implicit dependencies that aren't captured in their configuration. Imagine a new data serialisation format and a new data processing pipeline are each rolled out to 30% of users independently. The new pipeline was developed and tested *against* the new serialisation format-the developers assumed they'd be used together. But since the flags are independent, the probability that a user gets the new pipeline with the *old* serialisation format is:

$$P(\text{new pipeline} \cap \text{old serialisation}) = 0.30 \times 0.70 = 0.21$$

Twenty-one percent of your users are running an untested, potentially destructive combination that neither team knew to worry about. The flags interact destructively, but you won't discover this until production is on fire.

```
// Team A ships new serialisation. Team B ships new pipeline. Both at 30%.
// Nobody wrote down that B depends on A.


if (isFeatureEnabled(userId, 'new_serialisation', 30)) {
  writeDataInNewFormat(payload);   // writes JSON v2
}


if (isFeatureEnabled(userId, 'new_pipeline', 30)) {
  processWithNewPipeline(payload); // expects JSON v2 - but 70% of the
      time it gets v1
}
```

The two flags look independent in code review. The dependency only surfaces at runtime, for the 21% of users unlucky enough to hit the dangerous combination.

Temporal Non-Determinism Creates Race Conditions

A particularly insidious form of emergent non-determinism arises from the combination of distributed evaluation and dynamic configuration updates. Suppose a flag is evaluated at the beginning of a request in Service A, and then independently re-evaluated 200 milliseconds later in Service B. If a DevOps engineer updates the rollout percentage from 40% to 60% during that 200ms window, the two evaluations can produce different results for the same user on the same logical request.

This isn't a bug in either service in isolation-it's an **emergent property** of distributed evaluation combined with dynamic configuration. The result is authentication failures, data inconsistencies, and bugs that are completely unreproducible in any single-service environment.

Distributed Systems: The Perfect Storm for Non-Determinism

In distributed systems, each service becomes an independent source of non-determinism, and these sources compound multiplicatively.

The Fundamental Question: One Flag or Many?

When rolling out a feature across multiple services, you face a fundamental architectural choice: use a single shared flag across all services, or use per-service flags. For features that span multiple services, a single shared flag is almost always the safer design.

The reasoning becomes clear when you consider a three-service system where each service independently evaluates the same 25%-rollout flag. The probability that all three services agree on "new implementation" is only:

$$P(\text{all new}) = 0.25^3 = 1.5625\%$$

The probability that all three agree on "old implementation" is:

$$P(\text{all old}) = 0.75^3 = 42.1875\%$$

And therefore the probability that a user experiences *some mix* of old and new implementations across the three services is:

$$P(\text{mixed}) = 1 - 0.015625 - 0.421875 = 56.25\%$$

More than half your users are in a mixed state that almost certainly wasn't tested.

The Solution: Coordinated Flag Evaluation

The resolution is to evaluate flags *once* per request and then propagate the result. Rather than having each service independently call a flag evaluation function, flags are evaluated at the entry point of a request, stored in a distributed cache keyed by request ID, and then read-not re-evaluated-by downstream services. Every service that participates in handling a request sees the same flag values, eliminating cross-service inconsistency at the source.

Per-service flags remain appropriate when services are genuinely decoupled-a database migration in one service is truly unrelated to a UI change in another-but even then, explicit and unambiguous naming conventions are essential to prevent flags from being accidentally shared or reused.

Anti-Pattern #1: Nested Percentage Rollouts

One of the most common mistakes is nesting percentage-based flags-enabling a feature for some percentage of users, then enabling a sub-feature for some percentage of *those* users, and so on. The intuition feels natural, but the mathematics creates serious operational problems.

Compounding Probability

Consider a three-level nesting where 20% of users get a new experience, 30% of *those* users get an experimental checkout, and 50% of *those* users get one-click pay. The probability that any given user sees all three features is:

$$P(\text{all three}) = 0.20 \times 0.30 \times 0.50 = 0.03$$

Only 3% of users reach the deepest feature level. This creates several cascading problems.

Statistical Significance Collapse

If you're trying to measure whether one-click pay improves conversion rates, you're working with only 3% of your user base. The required sample size per variation for a valid A/B test is given by:

$$n = \frac{2(Z_{\alpha/2} + Z_{\beta})^2 \sigma^2}{\delta^2}$$

where $Z_{\alpha/2}$ is the critical value for your significance level, Z_{β} is the critical value for your desired power, σ^2 is the variance of the metric, and δ is the minimum detectable effect. For a typical scenario, detecting a 2% improvement in conversion rate with 95% confidence and 80% power, this formula yields approximately 4,700 users per variation.

At 3% of your user base, you'd need:

$$\text{Total daily users required} = \frac{4700}{0.03} \approx 156667$$

For smaller sites, this experiment can never reach significance. The feature is live in production, potentially affecting users, but you'll never be able to measure whether it's working.

Impossible States That Aren't Enforced

Nesting creates logical constraints between flags-a user can't have the experimental checkout without the new experience-but these constraints exist only in the minds of the engineers who wrote the code. The type system doesn't enforce them. If someone later refactors and removes the nesting structure, suddenly impossible states become possible. Users can end up in combinations that were never intended and never tested: one-click pay without the experimental checkout, or the experimental checkout without the new experience. The coupling is invisible until it breaks.

Unpredictable Configuration Changes

If a teammate updates the middle flag's percentage, the effect propagates through the entire nesting tree in ways that are non-obvious. Changing the checkout flag from 30% to 40% not only changes the checkout population-it changes the effective population for one-click pay, for any analytics measuring all three features, and for any capacity planning that assumed a 3% load from the deepest path. When these percentages are defined in separate configuration files, deployed by different teams at different times, the coupling is completely invisible until something breaks.

Anti-Pattern #2: Independent Evaluation Across Services

This anti-pattern is perhaps the most insidious because it looks correct at every local inspection. Two services implement what appears to be the same flag evaluation logic. But several forms of subtle divergence can cause them to disagree about which users are in which cohort.

Hash Function Divergence

If two services were written by different teams-or at different times-they may use different hash algorithms. The djb2 algorithm and a simple additive hash function produce completely different hash values for the same input.

```
// Order Service - djb2 hash (written by backend team)
function hash(str) {
  let h = 5381;
  for (const c of str) h = (h * 33) ^ c.charCodeAt(0);
  return Math.abs(h >>> 0);
}


// Email Service - additive hash (written by platform team, 6 months later
   )
```

```
function hash(str) {
  let h = 0;
  for (const c of str) h = ((h << 5) - h) + c.charCodeAt(0);
  return Math.abs(h);
}


// For userId='user_9182', featureName='loyalty_discount':
// Order Service  hash → bucket 73 → NOT in 25% rollout → no discount
    applied
// Email Service  hash → bucket 18 → IS  in 25% rollout → loyalty template
     sent
// Customer receives a loyalty email for an order that wasn't discounted.
```

String Concatenation Order

A subtler form of divergence is argument order in the hash input. Concatenating `userId + featureName` produces a completely different hash than `featureName + userId`. If someone "fixes" the order in one service for readability, they silently partition the user population differently, with no error messages and no obvious way to detect the problem.

```
// Service A  (original)
const key = userId + featureName;        // "user_9182loyalty_discount"


// Service B  ("fixed" for readability by a well-meaning engineer)
const key = featureName + userId;        // "loyalty_discountuser_9182"


// Completely different hash → completely different user cohort.
// No compiler warning. No runtime error. Just wrong users in wrong groups
    .
```

Mismatch Probability

When two services independently evaluate the same flag, the probability that a given user is *misclassified relative to the other service* is a function of the rollout percentage p:

$$P(\text{mismatch}) = 2p(1-p)$$

For a 25% rollout, this is $2 \times 0.25 \times 0.75 = 0.375$-a 37.5% chance of inconsistency for any given user. In the real-world incident described later, this produced a wave of confused customers contacting support because their discount email didn't match their cart total.

Anti-Pattern #3: Time-Based Non-Determinism

Time-based flags seem innocent-enabling a feature only during business hours is a completely reasonable thing to want. The danger is that time-based flags make flag evaluation a function of *when* something happens, and in distributed systems, different things happen at different times.

Intra-Request State Changes

Consider a user who starts filling out a form at 4:55 PM, when a business-hours flag is active, and submits the form at 5:02 PM, when the flag has deactivated.

```
// 4:55 PM - flag is active, advanced form is rendered with extra required
    fields
if (isBusinessHours()) {
  renderAdvancedForm(); // shows fields: address, VAT number, purchase
     order
}


// 5:02 PM - same user hits Submit; flag is now inactive
if (isBusinessHours()) {        // → false
  processAdvancedSubmission();  // never runs
} else {
  processBasicSubmission();     // runs - but the payload contains
     advanced fields it doesn't understand
}
// Result: validation failure, user's data is lost
```

The form was initialised with advanced fields and validation rules, but submission is processed with basic logic that doesn't understand those fields. Required data is missing. Validation fails. The user's work may be lost. This failure mode is completely invisible in development or staging environments, where you're unlikely to test across the boundary of a time-based flag.

Clock Skew in Distributed Systems

The problem compounds in distributed systems, where different servers may have different clocks. If Service A's clock is 30 seconds ahead and Service B's clock is accurate, they can simultaneously disagree about whether the current time is before or after the flag's boundary. The two services will evaluate the same time-based flag differently for the same request, producing exactly the kind of cross-service inconsistency described earlier-but now the root cause is the system clock rather than any visible code difference.

Anti-Pattern #4: Analytics Poisoning

When feature flag evaluation is repeated independently for analytics tracking, experimental results become invalid. A user might be rendered the new dashboard by one piece of code, but classified as a control-group user by the analytics tracking code-either due to different hash implementations, different timing, or different request contexts.

Compounding Misclassification

The problem compounds rapidly across service boundaries. If each of three services has only a 5% chance of misclassifying a given user's experiment group, the probability that a user is *correctly* classified in all three services is:

$$P(\text{all correct}) = (1 - 0.05)^3 = 0.95^3 \approx 0.857$$

$$P(\text{at least one wrong}) = 1 - 0.857 = 14.3\%$$

Nearly 15% of your analytics data is corrupted. You might conclude that a feature improves conversion rates when it actually doesn't-or miss a genuine improvement because the signal is buried in noise. Decisions get made on bad data, and the source of the corruption is completely non-obvious.

A Real-World Bug: The Disappearing Discount

Following production incident illustrates how quickly these anti-patterns cause real user harm. A loyalty discount flag ran at 25% rollout. The shopping cart service evaluated the flag to apply the discount. The email notification service independently evaluated the same flag to choose between a loyalty email template and a standard one.

Between cart calculation and email dispatch, a deployment updated the hash function implementation in the email service. The two services were now using different hash algorithms, so they produced different cohort assignments. Users who received the discount in their cart got the standard email-no mention of their discount. Users who didn't receive the discount got the loyalty email-with a discount amount that didn't match their actual order.

The mismatch probability for a 25% rollout is:

$$P(\text{mismatch}) = 2 \times 0.25 \times 0.75 = 0.375$$

Over a third of all users experienced some form of inconsistency. Customer support was flooded. The root cause took hours to identify because both services were individually behaving correctly-the bug only existed in the *relationship* between them.

Additional Distributed Systems Sources of Non-Determinism

Beyond flag implementation mistakes, distributed systems introduce further sources of non-determinism that interact with feature flags.

Network Partitions

When a service cannot reach the flag evaluation service, it must fall back to some default value. If different services in the same request fanout use different fallbacks due to different network conditions, they will disagree about flag values. The emergent non-determinism here is a function of network topology, not code-making it especially difficult to reproduce or diagnose.

Eventual Consistency

When flag configuration is updated in a central store and replicated to regional nodes, different services reading from different replicas may see different configuration states simultaneously. A service in US-East might read the updated 50% rollout while a service in EU-West still reads the old 20% rollout, because replication lag means the two regions are briefly in different states. For the duration of that lag, users whose requests are handled across regions will receive inconsistent flag values.

Deployment Skew

During a rolling deployment, old and new versions of a service run simultaneously. If the deployment includes a change to the hash function-perhaps for a bug fix or performance improvement-users whose requests are handled by the old version will receive different cohort assignments than users handled by the new version. This creates a window of inconsistency that persists for the duration of the deployment, which can be minutes or hours depending on cluster size.

Anti-Pattern Summary

The four anti-patterns described above share a common thread-they all introduce non-determinism at points where the system implicitly assumes consistency. The table below captures their distin-

guishing characteristics side by side, which is useful when auditing an existing codebase or reviewing a new flag implementation.

Anti-Pattern	Root Cause	Scope of Impact	Failure Mode	Detection Difficulty
Nested percentage rollouts	Multiplicative probability from chained flag evaluations	Exponential reduction in experiment population; implicit flag coupling	Invalid flag combinations become possible after refactoring; experiment never reaches statistical significance	Medium - visible in code review, but probability math is easy to overlook
Independent cross-service evaluation	Each service re-evaluates flags from raw user ID using potentially divergent logic	All users whose requests span multiple services	Mismatched UX across services (e.g. discount applied but not acknowledged in email); data corruption at service boundaries	High - both services appear correct in isolation; bug only exists in their relationship
Time-based non-determinism	Flag value changes mid-session due to clock boundaries or cross-service clock skew	Users whose sessions straddle a flag boundary; requests fanned out across nodes with clock drift	Form/submission mismatch; workflow initialised under one flag state, completed under another	High - only reproducible at exact clock boundaries; never manifests in dev environments
Analytics re-evaluation	Tracking code evaluates the same flag independently from rendering code	All experiment analytics; can corrupt entire A/B test datasets	Users miscategorised in experiment groups; false positive or negative experimental results; decisions made on invalid data	Very high - data looks plausible; corruption is silent and statistical

A useful heuristic: if you find yourself calling `isFeatureEnabled()` more than once for the same flag in the same logical flow-whether within a single service or across a request spanning multiple services-you are likely instantiating one of these anti-patterns.

The Safety Paradox Quantified

Let's put concrete numbers on how feature flags can increase rather than decrease risk. Without feature flags, you have a single code path, a single deployment configuration, and a single system state to test. Risk is concentrated in the deployment event itself.

With three non-deterministic feature flags at 50% rollout each, you have $2^3 = 8$ possible system states. If you then run three independent services that each evaluate flags independently, the number of possible states becomes:

$$\text{Total states} = (2^3)^3 = 2^9 = 512$$

Suppose your team thoroughly tests ten of these states. The probability that a given user request encounters an untested state is:

$$P(\text{untested}) = 1 - \frac{10}{512} \approx 98\%$$

Nearly every user is running untested code. Feature flags, intended to reduce risk by enabling gradual rollout, have instead ensured that virtually all production traffic runs through states that were never validated. This is the safety paradox in its starkest form.

The Solution: Centralized, Cached Flag Evaluation

The pattern that resolves these problems is straightforward in principle, though it requires discipline to implement consistently: **evaluate each flag once, cache the result, and propagate it explicitly**.

Core Principles

These four principles form the contract that the solution must uphold. They are worth making explicit because violating any one of them-even inadvertently, in a single service or code path-is sufficient to reintroduce the inconsistencies described above.

Principle 1 - Single Source of Truth: For any given user request, all feature flags are evaluated exactly once, at request inception, creating an immutable flag context that propagates to all services involved in handling the request. There is no legitimate reason for a downstream service to call the flag evaluation function with a user ID it already received; if it is doing so, it is re-deriving information that should have been passed to it.

Principle 2 - Temporal Consistency: Flag values remain constant for the duration of a user session or request, regardless of configuration updates, time boundaries, or service-to-service propagation delays. A user who began a checkout flow under a particular set of flag values must complete that flow under the same values, even if a configuration change is deployed mid-session. The flag context is evaluated once and frozen.

Principle 3 - Explicit Propagation: Flag evaluation results are explicitly passed between services as first-class request metadata, never re-evaluated from user identifiers. This makes the flow of flag state visible in the request structure itself-any service that receives a request can inspect the flags it is operating under without needing access to the flag evaluation service.

Principle 4 - Audit Trail: Every flag evaluation is logged with complete context including hash values, configuration versions, timestamps, and evaluation logic version, enabling post-hoc analysis of user experiences. When a production incident occurs, you must be able to answer the question "exactly which flag values did this user have at this moment?" without speculation.

Together these principles transform feature flags from a source of emergent non-determinism into a tractable, observable, and debuggable system property.

```
// Entry point: evaluate once, store against the request ID
async function handleRequest(req) {
  const flags = await flagService.evaluateAll(req.userId);
  await cache.set(`flags:${req.requestId}`, flags, { ttl: 3600 });
  req.flags = flags;
}


// Downstream services: read from cache, never re-evaluate
async function processPayment(requestId, order) {
  const flags = await cache.get(`flags:${requestId}`);  // same values,
     guaranteed
  return flags.newPaymentFlow
    ? chargeWithNewGateway(order)
    : chargeWithLegacyGateway(order);
}


async function sendConfirmation(requestId, order) {
  const flags = await cache.get(`flags:${requestId}`);  // still the same
     values
  return flags.newPaymentFlow
    ? renderNewTemplate(order)
    : renderLegacyTemplate(order);
}
```

At the entry point of every user request, all relevant flags are evaluated in a single operation and stored in a distributed cache keyed by request ID. Downstream services retrieve the cached result rather than

re-evaluating the flag. This guarantees consistency across all services for the lifetime of a request.

Every flag evaluation is logged with full context-user ID, session ID, flag values, configuration version, and timestamp. This makes production debugging tractable: you can reconstruct exactly what flag values a user had during any incident, rather than trying to reverse-engineer a hash function with a configuration that may have changed.

When flags have explicit dependencies-one-click pay requires the experimental checkout, the experimental checkout requires the new UI-those dependencies are validated at evaluation time. Any combination that violates a dependency is corrected and logged as a warning, rather than silently producing an invalid system state.

```
function validateFlags(flags) {
  if (flags.oneClick && !flags.newCheckout) {
    log.warn('oneClick requires newCheckout - disabling oneClick', flags);
    flags.oneClick = false;
  }
  if (flags.newCheckout && !flags.newUI) {
    log.warn('newCheckout requires newUI - disabling newCheckout', flags);
    flags.newCheckout = false;
    flags.oneClick    = false;
  }
  return flags;
}
```

Validation guards against configuration drift, flag system bugs, and the accidental removal of nesting guards during refactoring.

Handling Nested Features the Right Way: Mutually Exclusive Cohorts

Instead of nesting percentage-based flags, the cleaner approach is to define mutually exclusive cohorts using a single hash evaluation. Rather than having three independent flags with multiplicative interactions, you partition the hash space into explicit, named cohorts:

Cohort	Hash Range	Probability	Features
control	$[0, 70)$	70%	All old
variantA	$[70, 85)$	15%	New UI only
variantB	$[85, 95)$	10%	New UI + New Checkout
variantC	$[95, 100)$	5%	All new

This approach has several important advantages over nested flags. Population sizes are exact and pre-

dictable. All feature combinations are enumerated explicitly-there are no untested emergent states because every state is a named cohort. Statistical validity is maintained because sample sizes are known and fixed. Dependencies between features are enforced at evaluation time rather than implied by code structure. And analysis is straightforward: you're comparing four named groups rather than trying to reconstruct which combination of independent flags any given user had.

```
function assignCohort(userId) {
  const bucket = djb2Hash(userId + 'q3_experiment') % 100;


  if (bucket < 70) return { name: 'control',  newUI: false, newCheckout:
     false, oneClick: false };
  if (bucket < 85) return { name: 'variantA', newUI: true,  newCheckout:
     false, oneClick: false };
  if (bucket < 95) return { name: 'variantB', newUI: true,  newCheckout:
     true,  oneClick: false };
                   return { name: 'variantC', newUI: true,  newCheckout:
                      true,  oneClick: true  };
}
// Every possible feature combination is explicit and named.
// "newCheckout without newUI" cannot exist - it's not in the table.
```

Monitoring and Observability

Even with centralized evaluation and cohort-based assignment, production monitoring is essential. The actual distribution of users across cohorts should be sampled regularly and compared to the expected distribution. If a cohort shows more than 10% deviation from its expected population-which can happen due to hash function non-uniformity, unexpected user ID distributions, or configuration errors-an alert should fire before that deviation causes a production incident.

Cross-service consistency should also be monitored directly. By logging flag values alongside service calls and sampling recent requests, you can detect whether different services in the same request fanout are seeing different flag values. Any inconsistency here is a signal that the centralized evaluation pattern has been violated somewhere-a service is re-evaluating rather than reading from cache, or a fallback path is producing a different result than the primary path.

Key Takeaways

1. **Feature flags invert their safety promise when non-determinism emerges from flag interactions.** What was meant to reduce risk becomes a source of systemic risk.
2. **Emergent non-determinism arises from flag combinations, not individual flags.** Test the interaction space, not just individual flags in isolation.

3. **In distributed systems, use shared flags across services for coupled features.** Per-service flags create multiplicative complexity that grows as $2^{N \cdot S}$ where N is flags and S is services.
4. **Never evaluate the same percentage-based flag twice.** Evaluate once, cache the result, and propagate it through your system.
5. **Avoid nesting percentage-based flags.** Use mutually exclusive cohorts instead for cleaner probability math and statistical validity.
6. **Log everything.** Every flag evaluation should be logged with full context for debugging production issues.
7. **Validate dependencies.** Explicitly enforce relationships between feature flags to prevent invalid combinations from reaching production code.
8. **Monitor distributions and cross-service consistency.** Ensure your actual user distributions match expectations and that services agree on flag values.
9. **Centralize evaluation logic.** One hash function, one evaluation method, one source of truth prevents emergent divergence.
10. **Account for distributed systems properties.** Network partitions, eventual consistency, clock skew, and deployment skew all introduce additional non-determinism that compounds with flag complexity.

Conclusion

Feature flags are powerful tools, but non-deterministic flags introduce complexity that can silently corrupt data, poison analytics, and create baffling user experiences. The very mechanism designed to make deployments safer can make them more dangerous when emergent behaviours arise from flag interactions. Treat them with respect, implement proper safeguards, understand the emergent properties of your flag architecture, and always remember: every time you evaluate a percentage-based flag independently, you're rolling the dice on system consistency. And in distributed systems, those dice are already loaded.

References

[1] A. Tiwari, "Decoupling Deployment and Release- Feature Toggles," 2013, *Abhishek Tiwari*. doi: 10.59350/6ayra-sc056.