Feature Flags & Kill Switches

1. The Real Problem This Exists to Solve

Deploying new features directly to production is risky. Feature flags decouple deployment from release, allowing you to deploy code but control who sees it. Kill switches let you instantly disable features during incidents without redeploying. Together, they enable progressive rollouts, A/B testing, rapid incident response, and safe experimentation.

Real production scenario:

- New recommendation algorithm deployed to production
- **Without feature flags:**
  - Deploy new code
  - All users get new algorithm immediately
  - Bug discovered: recommendations show inappropriate content
  - Emergency: Must rollback entire deployment
  - Rollback takes 15 minutes (CI/CD pipeline)
  - 1 million users see inappropriate content
  - PR disaster, trust lost
  - Can't test incrementally
- **With feature flags:**
  - Deploy new algorithm (disabled by default)
  - Enable for 1% of users (canary)
  - Monitor metrics: CTR, engagement, errors
  - Everything looks good → increase to 10%
  - Bug discovered at 10%: inappropriate content
  - Flip kill switch → feature disabled in 2 seconds
  - Only 100,000 users affected (vs 1 million)
  - Fix bug, redeploy, re-enable gradually
  - Zero downtime, controlled rollout
  - Safe experimentation

The fundamental problem: Traditional deployments are all-or-nothing. Once code is deployed, all users see changes immediately. This makes experimentation dangerous, rollbacks slow, and incidents severe. Feature flags provide fine-grained control over who sees what, when, and enable instant rollback via configuration instead of redeployment.

Without feature flags:

- Risky all-or-nothing deploys
- Slow rollbacks (redeploy required)
- Can't test with real users incrementally
- Large blast radius for bugs
- Fear of shipping

With feature flags:

- Progressive rollouts (1% → 10% → 100%)
- Instant rollback (flip switch)
- A/B testing with real traffic
- Controlled experimentation

- Confidence in shipping

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Environment Variables (Requires Redeploy)

```
// Incorrect: Feature flag via environment variable
const NEW_ALGORITHM_ENABLED = process.env.NEW_ALGORITHM_ENABLED === 'true';

app.get('/api/recommendations', async (req, res) => {
  let recommendations;

  if (NEW_ALGORITHM_ENABLED) {
    recommendations = await getNewRecommendations(req.user.id);
  } else {
    recommendations = await getOldRecommendations(req.user.id);
  }

  res.json(recommendations);
});
```

Why it seems reasonable:

- Simple boolean check
- No external dependencies
- Easy to understand

How it breaks:

```
Problem 1: Requires redeploy to change
- Bug discovered in new algorithm
- Want to disable immediately
- Must change environment variable
- Restart all servers (5 minutes)
- During restart: service disruption
- Too slow for emergency

Problem 2: All-or-nothing
- Can't enable for 10% of users
- Can't A/B test
- Can't target specific users
- Binary: on or off for everyone

Problem 3: Different values per instance
- Server 1: NEW_ALGORITHM_ENABLED=true
- Server 2: NEW_ALGORITHM_ENABLED=false (forgot to update)
- Inconsistent behavior
- User gets different results depending on server
```

Production symptoms:

```
Incident response:
T0: Bug discovered
T1: Update environment variable
T2: Redeploy (5 minutes)
T5: Servers restarted
T7: Service restored

Timeline:
- 5+ minutes to disable feature
- Service disruption during restart
- Emergency rollback: slow and risky
```

❌ Incorrect Approach #2: Database Flag Without Caching (Performance Hit)

```
// Incorrect: Query database for every request
app.get('/api/recommendations', async (req, res) => {
  // Query DB on EVERY request
  const flag = await db.query(
    'SELECT enabled FROM feature_flags WHERE name = $1',
    ['new_algorithm']
  );

  const enabled = flag.rows[0].enabled;

  if (enabled) {
    recommendations = await getNewRecommendations(req.user.id);
  } else {
    recommendations = await getOldRecommendations(req.user.id);
  }

  res.json(recommendations);
});
```

Why it seems reasonable:

- Centralized flag storage
- Can update instantly
- No redeploy needed

How it breaks:

```
Performance impact:
- 10,000 requests/second
- Each queries feature_flags table
- 10,000 queries/second just for flags!
- Database overloaded
- Query latency: 50ms
- Total request latency: +50ms overhead
- P99 latency: 500ms (flag check alone)

Database load:
```

```
- Reads: 10,000/second (flags) + 5,000/second (actual data)
- Total: 15,000/second
- 67% of queries just for feature flags
- Wasted resources
- Connection pool exhaustion
```

Production symptoms:

```
Monitoring:
- Database CPU: 80% (was 40%)
- Top queries: SELECT * FROM feature_flags
- Query count: 3× increase
- Latency: +50ms on all endpoints

Cost:
- Database scaled up to handle load
- $2,000/month → $5,000/month
- Just for feature flag checks!
```

❌ Incorrect Approach #3: Code-Level Flags That Accumulate (Tech Debt)

```
// Incorrect: Flags never removed, code becomes spaghetti
app.get('/api/recommendations', async (req, res) => {
  // Flag from 2022 (2 years old!)
  if (FLAGS.oldAlgorithmV1) {
    recommendations = await getAlgorithmV1(req.user.id);
  }
  // Flag from 2023
  else if (FLAGS.oldAlgorithmV2) {
    recommendations = await getAlgorithmV2(req.user.id);
  }
  // Flag from 2024
  else if (FLAGS.newAlgorithmBeta) {
    recommendations = await getAlgorithmV3(req.user.id);
  }
  // Flag from 2025
  else if (FLAGS.mlAlgorithm) {
    recommendations = await getMlRecommendations(req.user.id);
  }
  // Current (2026)
  else {
    recommendations = await getCurrentAlgorithm(req.user.id);
  }

  res.json(recommendations);
});
```

Why it seems reasonable:

- Preserves history
- Can rollback to any version

- Incremental changes

How it breaks:

```
Technical debt:
- 5 code paths for same feature
- 4 old algorithms still in codebase
- getAlgorithmV1() hasn't been used in 2 years
- Nobody dares to remove old code (what if we need it?)
- Codebase bloat

Maintenance nightmare:
- Bug fix: Must update 5 different code paths
- Test coverage: Must test 5 algorithms
- Onboarding: New developers confused
- "Which algorithm is actually used?"
- "Can we delete this?"

Complexity:
- Cyclomatic complexity: 20+
- Untestable
- Fragile
- Fear of change
```

Production symptoms:

```
Codebase analysis:
- Lines of code: 50,000 → 75,000 (50% growth)
- Dead code: 15,000 lines (20%)
- Test failures: Flaky tests for old algorithms
- Build time: 15 minutes (was 5 minutes)

Developer survey:
- "I don't know which code is actually running"
- "Too scared to remove old flags"
- "Why do we have 5 algorithms?"
```

❌ Incorrect Approach #4: No Targeting (Can't A/B Test)

```
// Incorrect: Boolean flag, no user targeting
const featureFlags = {
  newAlgorithm: true, // On for everyone or no one
};

app.get('/api/recommendations', async (req, res) => {
  if (featureFlags.newAlgorithm) {
    // All users get new algorithm
    recommendations = await getNewRecommendations(req.user.id);
  } else {
    // All users get old algorithm
    recommendations = await getOldRecommendations(req.user.id);
```

```
  }

  res.json(recommendations);
});
```

Why it seems reasonable:

- Simple on/off switch
- Easy to implement
- Clear semantics

How it breaks:

```
Can't do progressive rollout:
- Want to test with 10% of users
- Flag is boolean: true or false
- Can't target specific percentage
- All or nothing

Can't do A/B testing:
- Want 50% on new, 50% on old
- Boolean flag doesn't support split
- Can't measure impact
- No control group

Can't target specific users:
- Want to enable for beta testers only
- Flag applies to everyone
- Can't whitelist users
- Can't test with internal team first
```

Production symptoms:

```
Product request:
PM: "Enable new algorithm for 10% of users"
Engineer: "Can't do that, flag is boolean"
PM: "Well, can you do 50/50 A/B test?"
Engineer: "Nope, all or nothing"
PM: "At least enable for internal team?"
Engineer: "Not with current implementation..."
```

❌ Incorrect Approach #5: No Kill Switch (Can't Emergency Disable)

```
// Incorrect: Feature fully deployed, no way to disable
app.get('/api/recommendations', async (req, res) => {
  // New algorithm always runs (no flag)
  const recommendations = await getNewRecommendations(req.user.id);

  res.json(recommendations);
});
```

Why it seems reasonable:

- Simpler code (no if statements)
- Feature is done, why need flag?
- Committed to new feature

How it breaks:

```
Incident:
T0: New algorithm deployed
T1: Bug discovered: Shows NSFW content to kids
T2: Need to disable immediately
T3: Only option: Rollback deployment
T4: CI/CD pipeline triggered
T5: Build: 3 minutes
T8: Deploy: 5 minutes
T13: Rollback complete

Problem:
- 13 minutes to disable feature
- Could have been instant with kill switch
- Affected users: 1,000,000 (13 min × 1,000 req/s)
- vs 2,000 users if instant disable (2 sec × 1,000 req/s)
```

Production symptoms:

```
Post-mortem:
"Root cause: No kill switch for new recommendation algorithm"
"Impact: 1M users saw inappropriate content"
"Time to mitigation: 13 minutes"
"Should have been: 2 seconds"
"Action item: Implement feature flags with kill switches"
```

3. Correct Mental Model (How It Actually Works)

Feature flags separate code deployment from feature release. Flags are stored in a fast, centralized system (database with cache, or Redis). Evaluation happens per-request with targeting rules (percentage, user ID, attributes). Kill switches are emergency flags that disable features instantly.

The Flag Evaluation Flow

```
1. Request arrives
2. Check cache for flag value (fast)
3. If not in cache, fetch from database
4. Evaluate targeting rules:
   - Percentage rollout (10%)
   - User attributes (beta_tester: true)
   - Override rules (user_id: 12345)
5. Return boolean: enabled or disabled
6. Execute code path based on result
7. Cache result for next request
```

Flag Types

```
Kill switch: Emergency disable (ignores all rules)
  - Priority: Highest
  - Example: disable_recommendations = true → OFF for everyone

Percentage rollout: Gradual enable
  - Example: 10% → random 10% of users see feature

User targeting: Specific users/attributes
  - Example: beta_testers, user_id in [1,2,3]

A/B test: Split traffic
  - Example: 50% variant A, 50% variant B
```

Lifecycle

```
Phase 1: Deploy code with flag (disabled)
Phase 2: Enable for internal team (testing)
Phase 3: Enable for 1% of users (canary)
Phase 4: Ramp to 10%, 25%, 50%, 100% (progressive)
Phase 5: Remove flag, clean up code (permanent)
```

4. Correct Design & Algorithm

Strategy 1: Redis-Backed Flags with Cache

```javascript
// Cache flag values in memory, refresh periodically
const flagCache = new Map();

setInterval(async () => {
  const flags = await redis.hgetall('feature_flags');
  for (const [key, value] of Object.entries(flags)) {
    flagCache.set(key, JSON.parse(value));
  }
}, 5000); // Refresh every 5 seconds
```

Strategy 2: Percentage-Based Rollout

```javascript
function isEnabled(userId: string, rolloutPercentage: number): boolean {
  const hash = hashCode(userId);
  return (hash % 100) < rolloutPercentage;
}
```

Strategy 3: Kill Switch with Priority

```
if (killSwitch.active) {
  return false; // Kill switch overrides everything
}

if (percentageRollout < 100) {
  return isInRollout(userId, percentageRollout);
}

return true;
```

5. Full Production-Grade Implementation

```
import Redis from 'ioredis';
import { createHash } from 'crypto';

/**
 * Feature flag configuration
 */
interface FlagConfig {
  enabled: boolean;
  rolloutPercentage: number;
  whitelist: string[];
  blacklist: string[];
  killSwitch: boolean;
  rules: FlagRule[];
}

interface FlagRule {
  attribute: string;
  operator: 'equals' | 'contains' | 'greaterThan';
  value: any;
}

/**
 * Feature flag service
 */
class FeatureFlagService {
  private redis: Redis;
  private cache = new Map<string, FlagConfig>();
  private cacheRefreshInterval = 5000; // 5 seconds

  constructor(redisUrl: string) {
    this.redis = new Redis(redisUrl);
    this.startCacheRefresh();
  }

  /**
   * Refresh cache periodically
   */
```

```
private startCacheRefresh() {
  setInterval(async () => {
    try {
      const flags = await this.redis.hgetall('feature_flags');

      for (const [flagName, configJson] of Object.entries(flags)) {
        const config = JSON.parse(configJson);
        this.cache.set(flagName, config);
      }

      console.log(`[Flags] Refreshed ${Object.keys(flags).length} flags`);
    } catch (error) {
      console.error('[Flags] Failed to refresh cache:', error);
    }
  }, this.cacheRefreshInterval);
}

/**
 * Check if flag is enabled for user
 */
isEnabled(
  flagName: string,
  context: {
    userId?: string;
    userAttributes?: Record<string, any>;
  } = {}
): boolean {
  const config = this.cache.get(flagName);

  if (!config) {
    console.warn(`[Flags] Unknown flag: ${flagName}, defaulting to false`);
    return false;
  }

  // Kill switch: instant disable
  if (config.killSwitch) {
    console.log(`[Flags] ${flagName} disabled by kill switch`);
    return false;
  }

  // Global enable/disable
  if (!config.enabled) {
    return false;
  }

  // Blacklist check
  if (context.userId && config.blacklist.includes(context.userId)) {
    return false;
  }

  // Whitelist check
  if (context.userId && config.whitelist.includes(context.userId)) {
```

```
      return true;
    }

    // Custom rules
    if (config.rules.length > 0 && context.userAttributes) {
      const rulesMatch = this.evaluateRules(config.rules, context.userAttributes);
      if (!rulesMatch) {
        return false;
      }
    }

    // Percentage rollout
    if (config.rolloutPercentage < 100 && context.userId) {
      return this.isInRollout(flagName, context.userId, config.rolloutPercentage);
    }

    return true;
  }

  /**
   * Evaluate targeting rules
   */
  private evaluateRules(rules: FlagRule[], attributes: Record<string, any>): boolean
{
    for (const rule of rules) {
      const value = attributes[rule.attribute];

      if (rule.operator === 'equals' && value !== rule.value) {
        return false;
      }

      if (rule.operator === 'contains' && !value?.includes(rule.value)) {
        return false;
      }

      if (rule.operator === 'greaterThan' && value <= rule.value) {
        return false;
      }
    }

    return true;
  }

  /**
   * Consistent hashing for percentage rollout
   */
  private isInRollout(flagName: string, userId: string, percentage: number): boolean
{
    const hash = createHash('md5')
      .update(`${flagName}:${userId}`)
      .digest('hex');
```

```
  const bucket = parseInt(hash.substring(0, 8), 16) % 100;

  return bucket < percentage;
}

/**
 * Set flag configuration
 */
async setFlag(flagName: string, config: FlagConfig): Promise<void> {
  await this.redis.hset('feature_flags', flagName, JSON.stringify(config));
  this.cache.set(flagName, config);
  console.log(`[Flags] Updated flag: ${flagName}`);
}

/**
 * Activate kill switch
 */
async killSwitch(flagName: string): Promise<void> {
  const config = this.cache.get(flagName) || {
    enabled: false,
    rolloutPercentage: 0,
    whitelist: [],
    blacklist: [],
    killSwitch: true,
    rules: [],
  };

  config.killSwitch = true;

  await this.setFlag(flagName, config);

  console.error(`[KILL SWITCH] ${flagName} disabled immediately`);
}

/**
 * Get all flags
 */
getAllFlags(): Map<string, FlagConfig> {
  return new Map(this.cache);
}

/**
 * Get flag metrics
 */
async getMetrics(flagName: string): Promise<any> {
  const enabledCount = await this.redis.get(`flag:${flagName}:enabled`);
  const disabledCount = await this.redis.get(`flag:${flagName}:disabled`);

  return {
    enabled: parseInt(enabledCount || '0'),
    disabled: parseInt(disabledCount || '0'),
    enabledPercent:
```

```
        (parseInt(enabledCount || '0') /
          (parseInt(enabledCount || '0') + parseInt(disabledCount || '0'))) *
        100,
    };
  }

  /**
   * Track flag usage
   */
  async trackUsage(flagName: string, enabled: boolean): Promise<void> {
    const key = `flag:${flagName}:${enabled ? 'enabled' : 'disabled'}`;
    await this.redis.incr(key);
  }
}

// Initialize
const featureFlags = new FeatureFlagService('redis://localhost:6379');

/**
 * Express middleware: Attach flags to request
 */
app.use((req, res, next) => {
  req.featureFlags = {
    isEnabled: (flagName: string) =>
      featureFlags.isEnabled(flagName, {
        userId: req.user?.id,
        userAttributes: {
          email: req.user?.email,
          plan: req.user?.plan,
          betaTester: req.user?.betaTester,
        },
      }),
  };

  next();
});

/**
 * Usage in endpoint
 */
app.get('/api/recommendations', async (req, res) => {
  try {
    let recommendations;

    if (req.featureFlags.isEnabled('new_recommendation_algorithm')) {
      console.log('[Feature] Using new recommendation algorithm');
      recommendations = await getNewRecommendations(req.user.id);

      // Track usage
      await featureFlags.trackUsage('new_recommendation_algorithm', true);
    } else {
      console.log('[Feature] Using old recommendation algorithm');
```

```
      recommendations = await getOldRecommendations(req.user.id);

      await featureFlags.trackUsage('new_recommendation_algorithm', false);
    }

    res.json(recommendations);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Admin endpoint: Update flag
 */
app.post('/admin/flags/:flagName', async (req, res) => {
  try {
    const { flagName } = req.params;
    const config: FlagConfig = req.body;

    await featureFlags.setFlag(flagName, config);

    res.json({ success: true, message: `Flag ${flagName} updated` });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Admin endpoint: Kill switch
 */
app.post('/admin/flags/:flagName/kill', async (req, res) => {
  try {
    const { flagName } = req.params;

    await featureFlags.killSwitch(flagName);

    res.json({ success: true, message: `Kill switch activated for ${flagName}` });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Admin endpoint: List flags
 */
app.get('/admin/flags', async (req, res) => {
  const flags = featureFlags.getAllFlags();

  const flagsArray = Array.from(flags.entries()).map(([name, config]) => ({
    name,
    ...config,
  }));
```

```
  res.json(flagsArray);
});

/**
 * Admin endpoint: Flag metrics
 */
app.get('/admin/flags/:flagName/metrics', async (req, res) => {
  try {
    const { flagName } = req.params;
    const metrics = await featureFlags.getMetrics(flagName);

    res.json(metrics);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Progressive rollout helper
 */
async function progressiveRollout(flagName: string) {
  const stages = [1, 5, 10, 25, 50, 100];

  for (const percentage of stages) {
    console.log(`[Rollout] Setting ${flagName} to ${percentage}%`);

    await featureFlags.setFlag(flagName, {
      enabled: true,
      rolloutPercentage: percentage,
      whitelist: [],
      blacklist: [],
      killSwitch: false,
      rules: [],
    });

    // Wait 1 hour between stages
    await new Promise(resolve => setTimeout(resolve, 3600000));

    // Check metrics
    const metrics = await featureFlags.getMetrics(flagName);
    console.log(`[Rollout] Metrics:`, metrics);

    // Could auto-rollback if error rate spikes
  }

  console.log(`[Rollout] ${flagName} fully deployed (100%)`);
}

/**
 * Example flag configurations
 */
```

```
async function setupInitialFlags() {
  // New recommendation algorithm (10% rollout)
  await featureFlags.setFlag('new_recommendation_algorithm', {
    enabled: true,
    rolloutPercentage: 10,
    whitelist: ['user_123', 'user_456'], // Beta testers
    blacklist: [],
    killSwitch: false,
    rules: [],
  });

  // Premium feature (only for paid users)
  await featureFlags.setFlag('advanced_analytics', {
    enabled: true,
    rolloutPercentage: 100,
    whitelist: [],
    blacklist: [],
    killSwitch: false,
    rules: [
      {
        attribute: 'plan',
        operator: 'equals',
        value: 'premium',
      },
    ],
  });

  // Beta feature (internal team only)
  await featureFlags.setFlag('experimental_feature', {
    enabled: true,
    rolloutPercentage: 100,
    whitelist: [],
    blacklist: [],
    killSwitch: false,
    rules: [
      {
        attribute: 'betaTester',
        operator: 'equals',
        value: true,
      },
    ],
  });
}

/**
 * React hook for client-side flags
 */
function useFeatureFlag(flagName: string): boolean {
  const [enabled, setEnabled] = useState(false);

  useEffect(() => {
    fetch(`/api/flags/${flagName}`)
```

```
      .then(r => r.json())
      .then(data => setEnabled(data.enabled));
  }, [flagName]);

  return enabled;
}

// Usage
function RecommendationsPage() {
  const newAlgorithm = useFeatureFlag('new_recommendation_algorithm');

  return (
    <div>
      {newAlgorithm ? (
        <NewRecommendations />
      ) : (
        <OldRecommendations />
      )}
    </div>
  );
}

/**
 * Flag cleanup (remove old flags)
 */
async function cleanupOldFlags() {
  const flags = featureFlags.getAllFlags();

  for (const [name, config] of flags.entries()) {
    // If flag is 100% rolled out and enabled
    if (config.rolloutPercentage === 100 && config.enabled && !config.killSwitch) {
      console.log(`[Cleanup] Flag ${name} is fully deployed, can be removed from
code`);

      // TODO: Create ticket to remove flag from codebase
    }
  }
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Progressive Rollout

```
// Day 1: 1% of users
await featureFlags.setFlag('new_feature', { rolloutPercentage: 1, ... });

// Day 3: 10% of users
await featureFlags.setFlag('new_feature', { rolloutPercentage: 10, ... });
```

```
// Day 7: 100% of users
await featureFlags.setFlag('new_feature', { rolloutPercentage: 100, ... });
```

Pattern 2: A/B Testing

```
const variant = req.featureFlags.isEnabled('new_design') ? 'B' : 'A';
trackMetrics(`variant_${variant}`, req.user.id);
```

Pattern 3: Emergency Kill Switch

```
// Incident detected
await featureFlags.killSwitch('problematic_feature');
// Feature disabled in 2 seconds (next cache refresh)
```

7. Failure Modes & Edge Cases

Cache Staleness

Problem: Flag updated, but cache not refreshed yet (5s delay).

Mitigation: Accept eventual consistency, or force cache refresh.

Redis Failure

Problem: Redis down, can't fetch flags.

Mitigation: Fallback to last known cache, default to safe value (disabled).

Flag Sprawl

Problem: 100+ flags, codebase spaghetti.

Mitigation: Regular cleanup, remove flags after 100% rollout.

8. Performance Characteristics & Tradeoffs

Flag Check Overhead

- **No cache:** 50ms (Redis query per request)
- **With cache:** 0.1ms (in-memory map lookup)

Rollout Time

- **Without flags:** 15 minutes (redeploy)
- **With flags:** 2 seconds (flip switch)

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Never Removing Old Flags

Fix: Clean up flags after 100% rollout and 2 weeks stable.

Mistake 2: No Kill Switch

Fix: Every flag should have emergency disable capability.

Mistake 3: Querying DB Per Request

Fix: Cache flags in memory, refresh every 5-10 seconds.

Mistake 4: Complex Nested Flags

Fix: Max 2 levels of nesting, prefer simple boolean checks.

Mistake 5: Client-Side Flags Expose Features

Fix: Server-side evaluation, client gets boolean result only.

10. When NOT to Use This (Anti-Patterns)

Configuration Values

Use config files, not feature flags (e.g., API keys, URLs).

Business Logic

Don't encode business rules in flags (use rules engine).

Permanent Toggles

If flag will be on forever, just deploy code directly.

11. Related Concepts (With Contrast)

A/B Testing

Related: Feature flags enable A/B tests (variant A vs B).

Canary Deployment

Related: Flags enable canary (1% rollout).

Circuit Breaker

Difference: Circuit breaker auto-disables on errors, kill switch is manual.

12. Production Readiness Checklist

Flag Service

- [ ] Redis-backed flag storage
- [ ] In-memory cache (refresh every 5s)
- [ ] Percentage-based rollout support
- [ ] User targeting (whitelist, rules)
- [ ] Kill switch capability

Integration

- [ ] Middleware attaches flags to request
- [ ] Simple API: req.featureFlags.isEnabled('flag_name')
- [ ] Track flag usage metrics
- [ ] Admin UI to manage flags

Operations

- [ ] Document all active flags
- [ ] Regular cleanup (remove old flags)
- [ ] Kill switch runbook
- [ ] Progressive rollout automation

Monitoring

- [ ] Track flag evaluation count
- [ ] Alert on kill switch activation
- [ ] Dashboard: rollout percentage over time
- [ ] Metrics: enabled vs disabled per flag