Heroku Migration Monitoring Checklist

Keep app visibility simple wherever your application lands next

Moving from Heroku to a new platform can give your team more flexibility, control, and pricing options. It can also give your team more production responsibility.

This checklist helps small engineering teams protect application visibility before, during, and after a Heroku migration. Use it whether you are evaluating AWS, DigitalOcean, Render, Fly.io, Railway, Kubernetes, ECS/Fargate, a VPS provider, a managed container platform, or something else entirely.

The goal is simple: keep your team from losing the app-level visibility Heroku made easy to ignore until something went wrong.

Who this checklist is for

Use this checklist if:

- You are moving a Rails, Ruby, Python, Django, Flask, Laravel, or Phoenix application off Heroku.
- You are still deciding which platform should replace Heroku.
- Your team does not have a dedicated DevOps, SRE, or platform engineering group.
- You need to compare app behavior before and after migration.
- You want to avoid turning monitoring into a separate infrastructure project.
- You need practical visibility into slow endpoints, database queries, background jobs, deploys, and errors.

The monitoring goal

The goal is not to monitor everything on day one. The goal is to keep enough application visibility to answer five questions quickly:

1. What got slower?
2. When did it get slower?
3. Which endpoint, job, query, or external call caused it?
4. Did the change happen during migration, after a deploy, or after a platform/infrastructure change?

5. What should the team fix first?

If your monitoring cannot answer those questions, the migration will create avoidable debugging work.

Platform-neutral migration principle

Every Heroku replacement creates a different operational tradeoff:

Destination type	What usually improves	What can get harder	Monitoring implication
AWS	Control, flexibility, procurement paths, service breadth	Infrastructure complexity and ownership	Keep app-level APM separate from infrastructure noise
DigitalOcean	Simpler cloud with managed databases, app platform, and Kubernetes	Less service breadth than AWS, different scaling model	Baseline before migration and compare managed-service performance
Render, Fly.io, Railway, or PaaS-like platforms	Developer experience and simpler deploys	Platform-specific limits and performance differences	Baseline app behavior before trusting the new defaults
Kubernetes or ECS/ Fargate	Portability, control, container-native workflows	Operational surface area	Make deploy markers, request traces, and job visibility non-negotiable
VPS or self-managed servers	Cost control and simplicity for some apps	More direct server responsibility	Watch app performance and system symptoms together

Quick-start checklist

Use this section if you only have 30 minutes.

Phase	Minimum action	Done
Before migration	Record your top 10 slowest endpoints in Heroku production	☐

Phase	Minimum action	Done
Before migration	Capture current error rate and top recurring exceptions	☐
Before migration	Identify your highest-volume database queries	☐
Before migration	List critical background jobs and queue expectations	☐
During migration	Compare response times between Heroku and the new platform	☐
During migration	Confirm deploy markers are visible in monitoring	☐
During migration	Watch database latency and query volume	☐
After migration	Review the top slow endpoints weekly for 30 days	☐
After migration	Tune alerts so developers can actually respond	☐
After migration	Document the “what changed?” debugging workflow	☐

Phase 1: Before migration

Establish a performance baseline

Before changing platforms, capture how the app behaves today. This gives your team a reference point when something feels slower after the move.

What to baseline	Why it matters	Done
Top 10 slowest web endpoints	Shows which user-facing actions are already risky	☐
Top 10 highest-traffic endpoints	Prevents you from over-focusing on obscure slow paths	☐
Average and p95 response time for key requests	Gives a realistic before/after comparison	☐

What to baseline	Why it matters	Done
Current error rate	Helps separate migration issues from existing noise	☐
Most frequent exceptions	Identifies problems that should not be blamed on the new platform	☐
Current database response time	Creates a baseline for post-migration query behavior	☐
Queue latency and job duration	Helps catch background processing regressions	☐
External API call timing	Helps identify network or integration changes later	☐

Identify business-critical paths

Not every endpoint deserves equal attention. Choose the paths that matter most to customers, revenue, operations, or internal users.

Examples:

- Signup
- Login
- Checkout or billing
- Search
- Dashboard load
- Report generation
- API endpoints used by customers
- Admin workflows used by support or operations
- Background jobs that send emails, process payments, sync data, or generate reports

Find database risks before they become platform mysteries

Database problems are easy to misdiagnose during migration. A slow query that existed on Heroku can look like a problem with the new platform if you did not capture it first.

Check for:

- N+1 queries
- Slow SQL queries
- Missing or unused indexes
- High query counts per request

- Long-running transactions
- Queries that behave differently under production data volume
- Background jobs that create database pressure

Confirm observability basics

Before you migrate, make sure the app can report the signals your team will need during the move.

Signal	Why it matters	Done
Application performance traces	Shows where time is spent in the code path	☐
Error tracking	Connects exceptions to real production behavior	☐
Deploy markers	Helps correlate regressions with releases	☐
Logs linked to requests or errors	Shortens debugging time	☐
Background job timing	Captures async work that users still depend on	☐
Database query timing	Separates app slowness from data-layer slowness	☐
Alert routing	Makes sure the right person sees important issues	☐

Phase 2: While choosing the new platform

Evaluate monitoring impact before choosing

Platform choice is not only about hosting features. It also changes how your team will debug production.

Evaluation question	Why it matters
Can we install APM easily?	The app team needs visibility before cutover
Can we see deploy markers?	Regressions need a timeline
Can we monitor background jobs?	Async work often breaks quietly
	Migration needs before/after evidence

Evaluation question	Why it matters
Can we compare Heroku vs. the new platform?	
Can developers access the data directly?	Small teams cannot route every issue through ops
Will alerts be actionable?	Noisy alerts will be ignored
Does the buying path fit our team?	Procurement can block technically good choices

Phase 3: During migration

Compare Heroku and the new platform

During the migration window, compare behavior across the old and new environments. The goal is not perfect parity. The goal is to catch meaningful differences before the cutover becomes permanent.

Compare	What to watch	Done
Response time	Are key endpoints faster, slower, or more variable?	☐
Error rate	Did new errors appear on the new platform?	☐
Database timing	Did query latency change?	☐
Query volume	Are requests making more database calls than before?	☐
Queue latency	Are background jobs keeping up?	☐
External calls	Did API or service timing change?	☐
Memory symptoms	Did new memory pressure appear?	☐
CPU symptoms	Are requests or jobs more compute-bound?	☐

Watch the migration danger zones

These are the areas most likely to create confusing performance changes during a Heroku migration.

Danger zone	Symptom	Monitoring question
Database connection settings	Timeouts, slow requests, queue buildup	Are requests waiting on database access?
Network path changes	External calls become slower	Which service calls changed after the move?
Background workers	Jobs lag or fail silently	Are queues growing or job durations increasing?
Environment variables	New errors after deploy	Did configuration drift create exceptions?
Asset or file handling	Pages load slowly or fail	Are requests waiting on storage or asset paths?
DNS and routing	Intermittent failures	Are errors clustered around cutover timing?
Autoscaling assumptions	Performance changes under load	Are slowdowns tied to traffic spikes?

Phase 4: Cutover day

Cutover monitoring checklist

On cutover day, your team should have one person watching application behavior, not just platform health.

Check	Good sign	Done
Homepage or primary app entry point	Loads successfully and within expected range	☐
Login/session flow	No new errors or latency spikes	☐
Key customer workflow	Response time comparable to baseline	☐
API endpoints	No new timeout or authentication failures	☐

Check	Good sign	Done
Database queries	No unexpected slow query spike	☐
Background jobs	Queues drain as expected	☐
Error rate	No sustained increase from baseline	☐
Logs	No new recurring exception pattern	☐
Deploy marker	Cutover is visible on timeline	☐

Cutover triage questions

If something goes wrong, start with these questions:

1. Is the issue affecting all traffic or one endpoint?
2. Did the issue start exactly at cutover or after a deploy?
3. Is the app waiting on the database?
4. Is the app waiting on an external service?
5. Are background jobs failing or backing up?
6. Is the error new, or did it exist before migration?
7. Is this a code path issue, configuration issue, or platform issue?
8. Can the team roll back, route around it, or fix forward?

Phase 5: First 30 days after migration

Weekly performance review

For the first month after migration, hold a short weekly review. Keep it tactical. The goal is to spot regressions and improve the monitoring setup while the migration is still fresh.

Weekly question
What were the slowest endpoints this week?
Which endpoint changed the most from baseline?
Which database query caused the most pain?
Which background job created the most delay?
Which errors were noisy but low-impact?

Weekly question
Which errors affected customers?
Did any deploy cause a visible regression?
What is the one performance fix to prioritize?

Tune alerts for a small team

Small teams cannot respond to every noisy signal. Alerts should point to issues that need human attention.

Use alerts for:

- Sustained error-rate increases
- Critical endpoint latency spikes
- Background queue delays that affect users
- Database latency that persists beyond a short spike
- Job failures that block customer workflows
- Regressions after deploys

Avoid alerts for:

- One-off slow requests
- Known noisy exceptions that do not affect users
- Metrics no one owns
- Thresholds copied from another company's environment
- Alerts that wake people up without a clear action

Document your debugging workflow

After the migration, write down the workflow your team will use when someone says, "The app is slow."

1. Check whether the issue is isolated or widespread.
2. Look at the affected endpoint, job, or API path.
3. Compare timing to the pre-migration baseline.
4. Check deploy markers and recent platform changes.
5. Inspect database queries and external calls.
6. Review related errors and logs.
7. Decide whether the next action is code, database, configuration, or infrastructure.
8. Record the fix so the next incident is faster.

Monitoring readiness scorecard

Score each item from 0 to 2. (0 = Not in place, 1 = Partially in place, 2 = Ready)

Category	Question	Score
Baseline	Do you know your current slowest endpoints?	
Baseline	Do you know your current error rate?	
Baseline	Do you know your current database bottlenecks?	
Coverage	Can you trace a slow request to the code path?	
Coverage	Can you see database query timing?	
Coverage	Can you monitor background jobs?	
Timeline	Can you see deploy markers?	
Timeline	Can you distinguish app deploys from platform changes?	
Ownership	Does every important alert have an owner?	
Ownership	Does the team know what to do when an alert fires?	
Cutover	Do you have a cutover-day monitoring owner?	
Cutover	Do you have a rollback or fix-forward plan?	

0-8: High risk. Baseline the app before migration and prioritize request traces, errors, database timing, and deploy markers.

9-16: Partially ready. Focus on the missing categories before cutover.

17-24: Ready enough to move carefully. Keep reviewing performance weekly for the first 30 days after migration.
