Taking a step back is often necessary

We needed a second background check provider in a system built around one. Refactoring first took longer than the feature itself. After that, adding new providers became trivial.

Igor Leal Antunes · LinkedIn · Published Nov 11, 2026 · Updated Nov 11, 2026

> **The takeaway**
>
> We were asked to add a second background check provider, but the system was hardcoded to the first one. Instead of patching the new provider in, we stopped and refactored first, even though that step changed nothing users could see and took longer than the feature itself. Once the refactor was in, the actual work was the easy part.

The request sounded small: let employers order background checks from a second provider. Our ATS (applicant tracking system) already had background checks. An employer picks a package on the hiring screen, the candidate gets an invitation, the provider sends results back through webhooks, and onboarding waits until the check is done. Adding a second option to that list looked like a quick job.

Then I opened the code. The integration had run for about a decade and did its job well, but it was built for one provider because there was only ever one. The system had never needed to ask which provider it was talking to, so there was no place to plug in another.

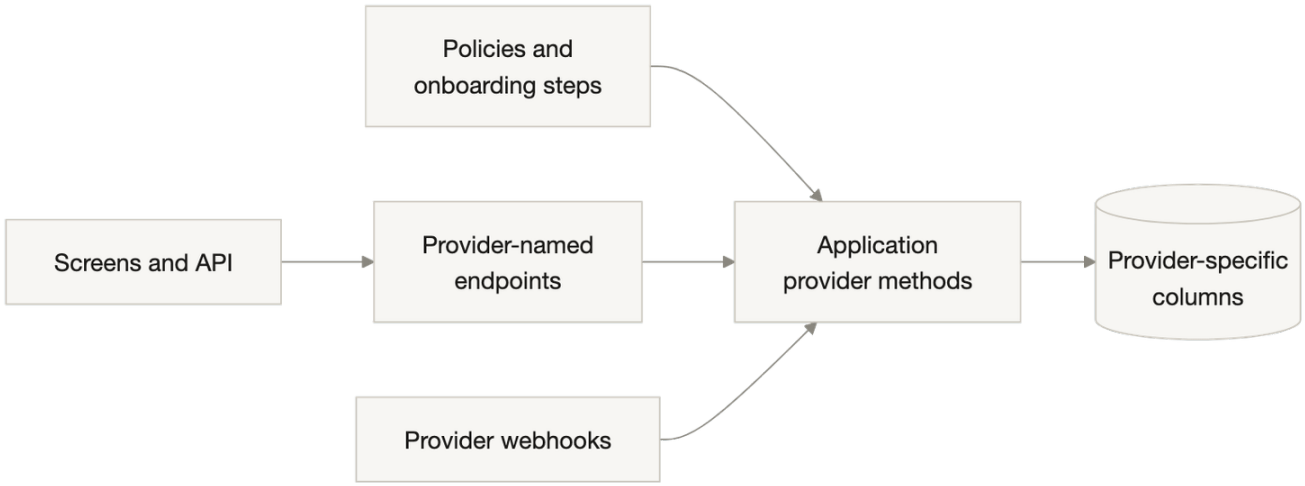
We had to take a step back before adding the new provider, and put an abstraction layer between the existing calls and the provider behind them. That meant a refactor, a data migration and a careful rollout before any new provider code. Adding the providers afterwards was a breeze, and that part is short.

Hardcoded to one provider, everywhere

The original provider wasn't behind an integration layer. It was hardcoded through the whole codebase, starting with `Application`, the main model of the ATS, which ties a candidate to a job:

- provider-specific columns on the application table, holding the candidate id, report id, package and status;
- eight methods hardcoded to the provider on the `Application` model, such as `getProviderStatus()` and `getProviderReportRelation()`, so every caller had to know which provider it was dealing with;
- provider-named endpoints, controllers and request classes;
- permission rules and onboarding steps that checked for that provider by name;
- API responses with a provider-named field that the UI read directly.

The columns weren't a great database design. They look like something added in a hurry to ship the first integration, and they worked well enough that nobody had a reason to revisit them.



Before: every path ends in provider-specific code and storage. A second provider would have needed a second copy of each box.

The quick version was an `if` in each of those places. It would have worked, and the third provider would have added another `if` next to it. So we split the work in two: first make the system provider-agnostic with the original provider still the only one, then add the new providers on top.

Two interfaces and a resolver

The abstraction layer has three parts:

- `BackgroundCheckable`: the thing being checked. `Application` implements it.
- `BackgroundCheckProvider`: a company that runs checks. Each provider is one class.
- The resolver: knows which providers exist and which ones a company can use, so the rest of the app never has to.

`BackgroundCheckable` is small on purpose. It's what a provider needs to know about the candidate, and nothing about how any provider works:

```
interface BackgroundCheckable
{
    public function backgroundChecks(): MorphMany;

    public function backgroundCheckLocation(): Location;

    // The id we send to the provider, so its webhooks can find us again.
    public function backgroundCheckReference(): string;

    // ...
}
```

The provider contract covers the lifecycle of a check: is it turned on for this company, what can be ordered, place the order, read the result.

```
interface BackgroundCheckProvider
{
    public function reference(): string;

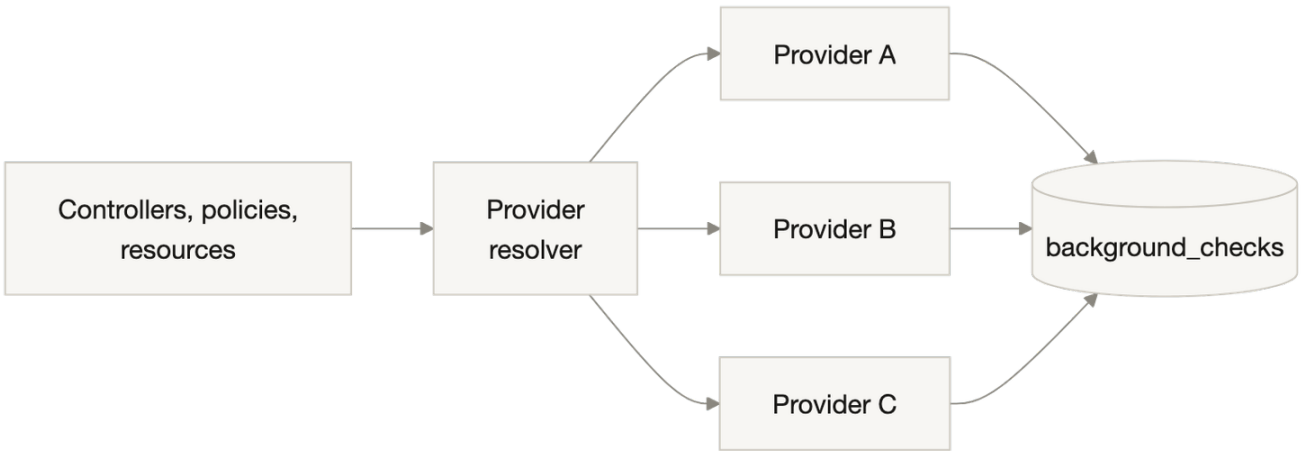
    public function isEnabledFor(Company $company): bool;

    public function packages(Company $company): Collection;

    public function order(BackgroundCheckable $checkable, PackageSelection $selection): Invitation;

    public function result(BackgroundCheckable $checkable): BackgroundCheckResult;

    // ...
}
```



After: callers only know the resolver and the interfaces. Each provider is a class, and they all share one table.

At first the resolver's list held exactly one class: the original provider. Every screen, endpoint and permission check now went through the abstraction and ended up at the same provider as before. Lots of new code, same behavior. That was the goal.

Rewiring the call sites, one by one

With the interfaces in place, most of the work was replacing direct calls. Here's a simplified version of one of them, the check for whether a candidate's background check is still in progress:

```php
// Before
$candidateId = $application->getIntegrationRelation(ProviderService::ID, 'candidate');

if ($candidateId !== null) {
    $status = $application->getIntegrationRelation(ProviderService::ID, 'applicant_status');

    if ($status !== null) {
        $status = decrypt($status);

        if (!in_array($status, ['clear', 'consider'])) {
            return !in_array($status, ['pre_adverse_action', 'post_adverse_action']);
        }

        return false;
    }

    return true; // candidate invited, no status stored yet
}

return false;

// After
return $this->resolver
    ->resultsFor($application)
    ->contains(fn (BackgroundCheckResult $result) => $result->isPending());
```

The "after" doesn't know which provider ran the check, how it stores its data, or what its status strings are. Each provider maps its own statuses to our shared result type. It also changes what future work looks like: a new provider adds new code instead of editing existing code, so each change has far fewer places to introduce a bug.

Provider-named endpoints became generic ones that take a provider reference, and the API responses return a list of checks instead of a provider-named field. The first merge request touched 133 files. That's a lot to review, so it had one rule: behavior stays the same. Any diff that changed what a user would see was a bug in the refactor.

A new home for the data

The provider-specific columns had to go too, since a new provider would have needed its own set. Checks moved to a single `background_checks` table that any provider can use. It has a polymorphic relationship to its owner, so it isn't tied to `Application` either:

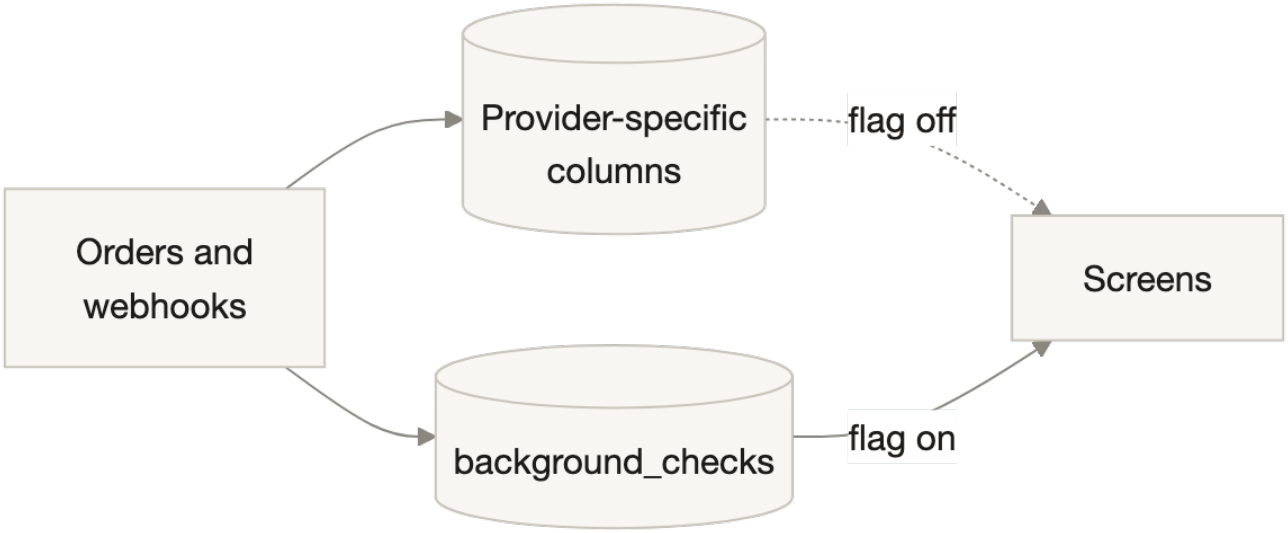
The main columns of background_checks.

Column	What it holds
checkable_type, checkable_id	The owner of the check, any BackgroundCheckable
provider	Which provider ran the check
external_id	The provider's id for the check, so webhooks can find the row
status, result	Our lifecycle and the outcome, encrypted at rest
idempotency_key	Unique, so the same event can't create a second row
metadata	Anything provider-specific, as encrypted JSON

Moving live data that onboarding depends on was the riskiest step, so it went behind a feature flag. The provider checked the flag on every read:

```
public function result(BackgroundCheckable $checkable): BackgroundCheckResult
{
    return Feature::active('background-checks-abstraction')
        ? $this->resultFromTable($checkable)
        : $this->resultFromLegacyColumns($checkable);
}
```

During the transition, every write went to both the old columns and the new table. We used dual writes to keep the old path available during the backfill: the old columns stayed complete, so turning the flag off went back to the previous behavior with nothing lost. With a large dataset that matters, because a backfill can take hours and the old path has to stay correct while it runs.

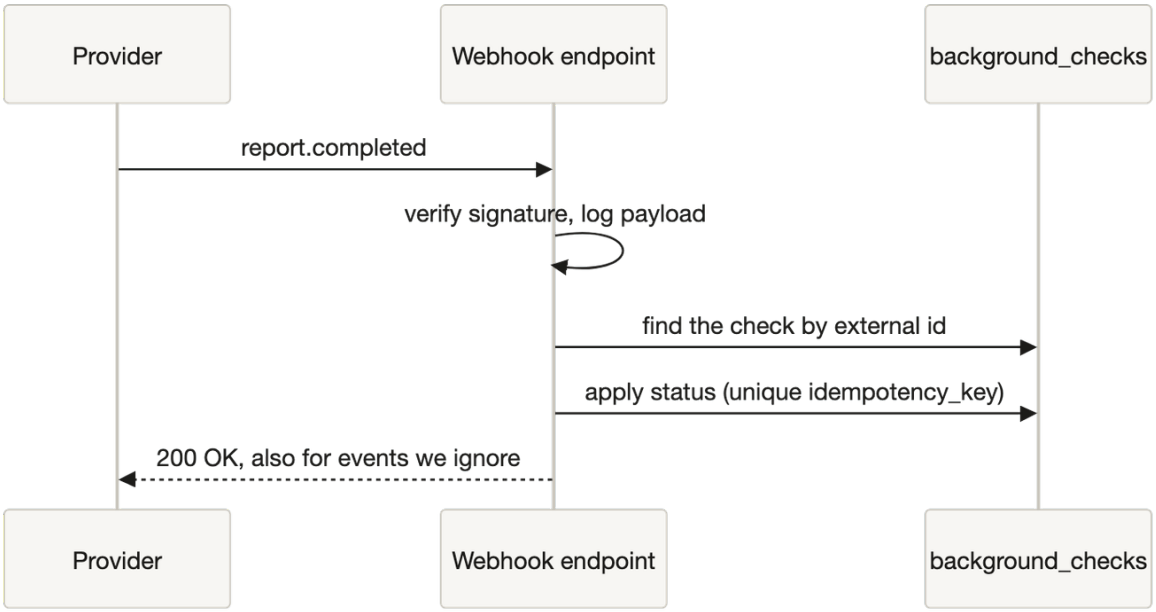


New writes reach both stores, and the backfill copies the history. Reads follow the flag.

A one-off backfill command copied the existing checks into the new table, while reads still used the old columns.

Webhooks arrive twice, late and out of order

Results come back through webhooks, and webhooks rarely arrive the way the documentation diagram suggests. The same event can be delivered twice, a `report.completed` can arrive before our own request has saved the order, and an event can show up for a candidate we've never seen. Stripe covers the same problems in its posts on idempotency and handling duplicate webhook events, and the same rules apply here.



A delivery that arrives twice finds the same row and applies the same status again.

The handler follows a few rules:

- Verify the signature before trusting the payload.
- Log every payload before doing anything with it. When a result looks wrong, the raw event is the first thing to check.
- Make every write idempotent. Each check has a unique `idempotency_key`, so a repeated delivery finds the existing check instead of creating a second one.
- Never let an older event overwrite a newer status.
- Return 200 for events we don't track, so the provider stops retrying them.
- Never call the provider's API from list pages. Webhooks write the state, and the screens read it.

Shipping took more planning than writing

The code went out in phases, with a way back from each one. This phase was all about the stability of the current system: if we could get past it, the rest of the work would be easy. We run on Kubernetes with rolling deploys, so during each rollout old and new pods serve traffic side by side, and every change has to work with both.

1. Run the migration that creates `background_checks` before any new pod takes traffic, because the new code eager-loads the relation on list pages.
2. Deploy the abstraction with the flag off. Every write now goes to both stores, and reads still use the old columns.

3. Watch errors and logs, and check that browser tabs opened before the deploy still work against the new API.
4. Run the backfill as a dry run in production. It reports how many rows it would copy and how many it can't map. Fix those rows.
5. Run the backfill for real, and check that the new table matches the old columns.
6. Turn the flag on to switch reads to the new table. The flag is stored in Redis, so it switches without a deploy or a restart.
7. Watch new orders and incoming webhooks until the numbers look normal.
8. Remove the flag and the old code.

Each phase and our response if it went wrong.

Phase	What could go wrong	Response
Deployed, flag off	A refactor bug in a code path we missed	Roll back the release
Backfill	A row that can't be mapped	Count it, report it, fix it and rerun
Flag on	Something nobody thought of	Flag off: the old columns are still complete

The riskiest moment was switching reads. Because the backfill had run and been checked first, and the old columns kept receiving every write, turning the flag off stayed a complete way back. Coordination mattered as much as the code: everyone knew which phase we were in, which screens to watch, and who would flip the flag.

Finally, adding the new providers was a breeze

After all of that (two interfaces, a new table, a feature flag, a runbook and a migration nobody was supposed to notice), we finally got to the part someone had actually asked for. We added the provider from the original request this way, and a third one soon after. For each one the work was:

1. Write a class that implements `BackgroundCheckProvider`.
2. Add it to the resolver's list.
3. Add a company setting that turns it on.

4. Write its webhook endpoint, which finds the check and applies the status, following the rules above.

The simpler of the two is about 200 lines. The hiring screens, the permission checks and onboarding didn't need a single new `if`, because they already asked the resolver. Most of the effort for the other one went into understanding their API and their order lifecycle, which is exactly where it should go.

Asking for time to change nothing

The hardest part of this project wasn't technical. The cleanup took far longer than adding a provider, and if it went well, users would notice nothing. It also carried real risk, because it moved live data that onboarding depends on. That's a hard thing to ask for. I had to talk it through with the people who set priorities and explain why the new provider would come later than a quick version would, and why waiting was the cheaper option.

The quick version would have been a second provider glued onto code built for one, then a third, with the branches multiplying in every controller and permission check. Each change would have been fast, and the system as a whole would have become very hard to change. The two providers that followed were the test of that decision, and neither needed a change to the hiring screens.

Once the new table had proven stable, the flag, the old columns' code and the transitional classes came out. Deleting code is one of the more pleasant parts of this job.

If I did it again, I'd write the provider interface even earlier, before the first refactor commit, and review it with the team on its own. Most of the later decisions followed from it.

Read online: https://igorlealantunes.com/articles/taking-a-step-back-is-often-necessary