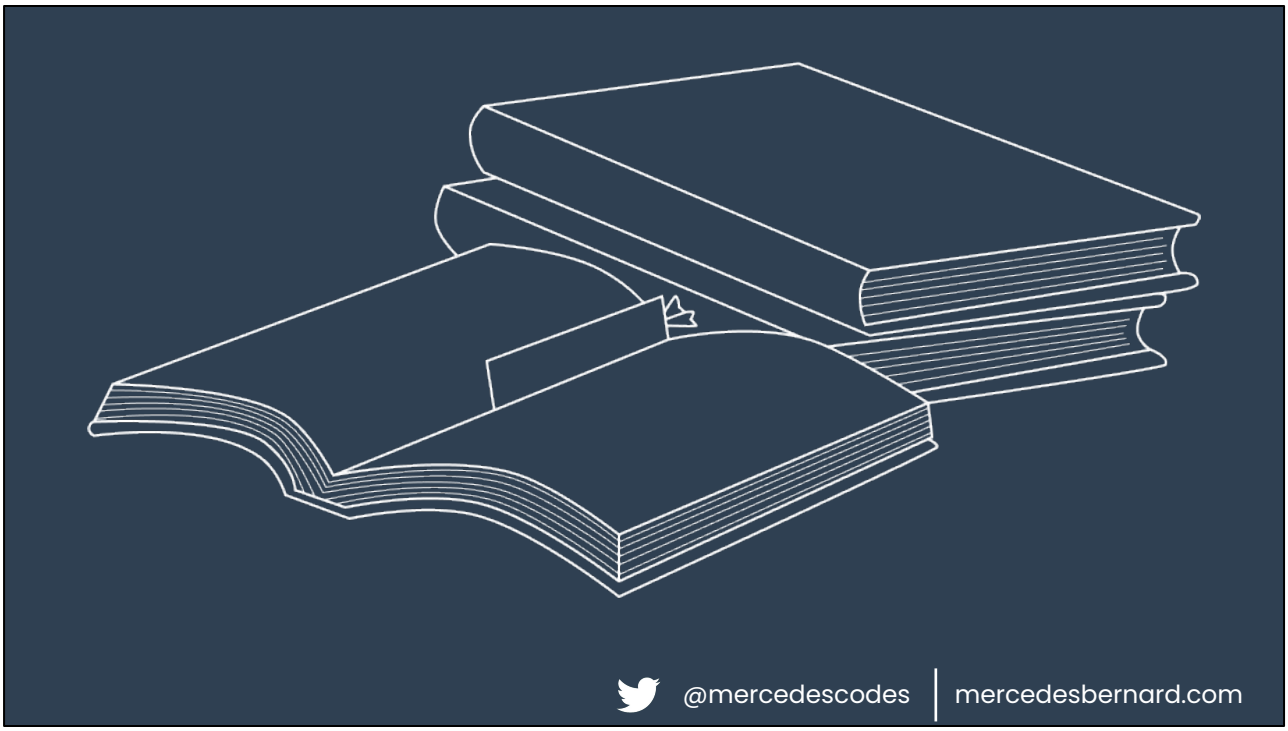
Hello! Today we're going to talk about how to assess legacy Rails projects that are in rough shape and figure out if we should save them and how to do it. This talk is written to be accessible for anyone who is faced with legacy code and wants to figure out incremental steps to refactor it into a delight to work with. There may be some things you don't understand if you're really early in your career but most of what we talk about will be useful, regardless of your level.

My name is Mercedes Bernard. My pronouns are she/her. And I'm the VP of Delivery at a digital consultancy in Chicago called Tandem.

If you are someone who likes to follow along with slides, I've put them on my website. The speaker notes will have everything I'm saying if that's helpful to you, and it might make it easier to follow some of the links and tools I'll share.



I had a recent client bring me 4 code repositories that were all in rough shape. They were undocumented and couldn't reliably be set up in a local dev environment. They were not containerized. There were multiple required dependencies that were not managed using any sort of package manager and had to be installed and configured manually. The reference data and seeds were out of date and threw lots of errors when you tried to run them. And there were rake tasks sprinkled all over the place for adding other reference data that you couldn't tell if you were supposed to use them or if they had been put there for one-off use cases.

None of the repos had any CI or CD. There was no documented way to deploy them. And any time a change was made, a lot of unintended side effects were introduced so the client lacked trust in any change or deployment and tried to avoid them at all costs.

2 of these repos were core to their business and had many users onboarded. They were generating revenue consistently. The other 2 weren't as vital and had started as proofs of concept.

The client wanted these 4 repositories stabilized so that they could start adding new features and continue building off what they had. Their main concern was cost. They were a startup with a short runway that was trying to raise another round of funding, and they didn't have a lot of money to invest so they wanted to find the most cost effective way to get their system in a place where they could market it to investors and continue to grow their business.

We had to decide whether we could salvage these projects or if we would have to tell

them that they weren't worth saving and should be scrapped. And for what we could salvage, we had to figure out the best way to do that–balancing both cost and long-term stability.

This is a practical, not philosophical talk. We're going to focus on how to make these decisions and how to complete rescue missions. All of the ideas we'll cover are relevant to any tech stack but the tools I'll share are specifically for Rails projects.

Gain an *empathetic* understanding

@mercedescodes | mercedesbernard.com

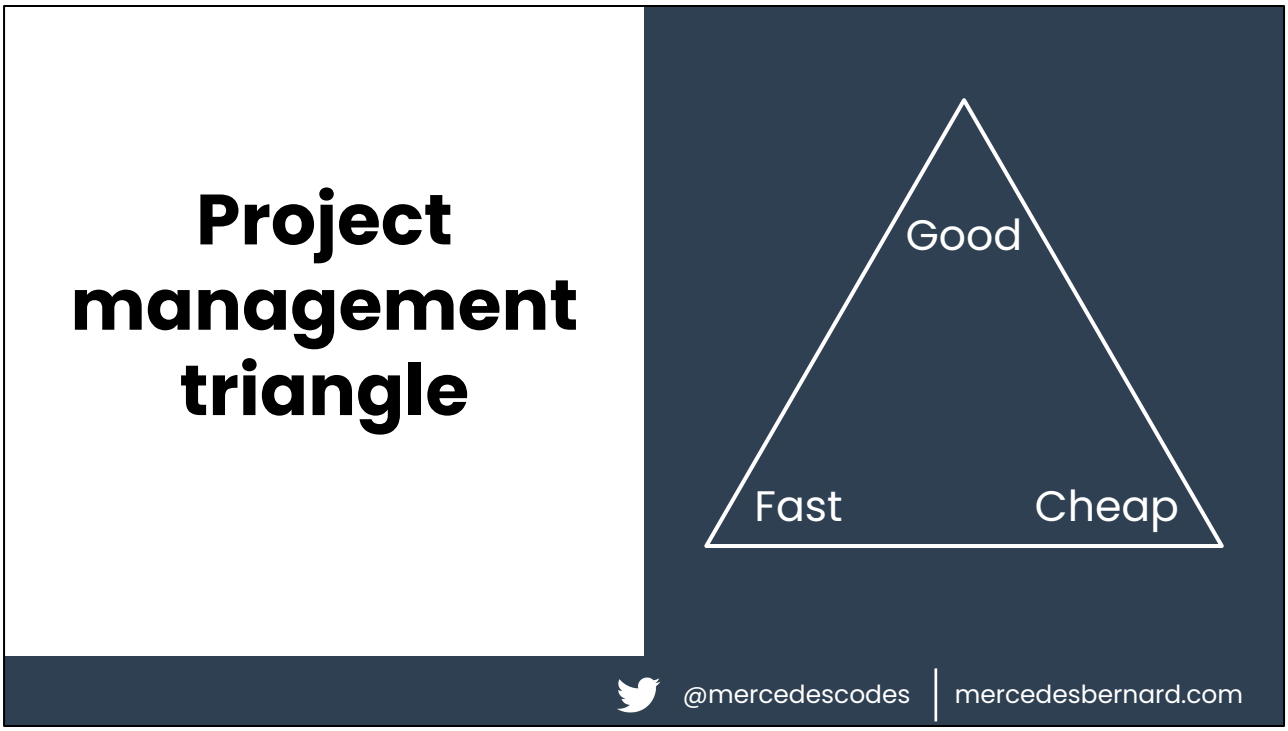
If you are given a rescue mission and are trying to figure out “should I stabilize this codebase or should I start over,” the best thing you can do is try to understand the context and current state of the system. It’s impossible to find the best solution forward without as full an understanding of the problem as you can get.

Codebases never become *legacy* through intentional bad decisions. There are always constraints and limitations placed on the humans who were writing the code before you, that you might not know about.

To build your empathetic understanding of what factors led to the current state of the project, take time to talk to stakeholders to understand the business and finances and how they have or have not changed over time.

- Maybe the company needed to invest more in marketing to try to find customers but that meant that they couldn't afford as large of a team to work on the code as they needed.
- Maybe they had to reduce the size of the team and layoffs meant that a lot of the knowledge left with those folks.
- Maybe they could only afford to hire contractors who didn't thoroughly understand the business.
- Maybe their team was made up of early-career devs who didn't have a lot of experience in the tech stack and had no mentorship to guide them so all of their choices were the best they could make with the information they had available.
- Maybe the business went in a new direction and everyone is still learning about the new position and industry.

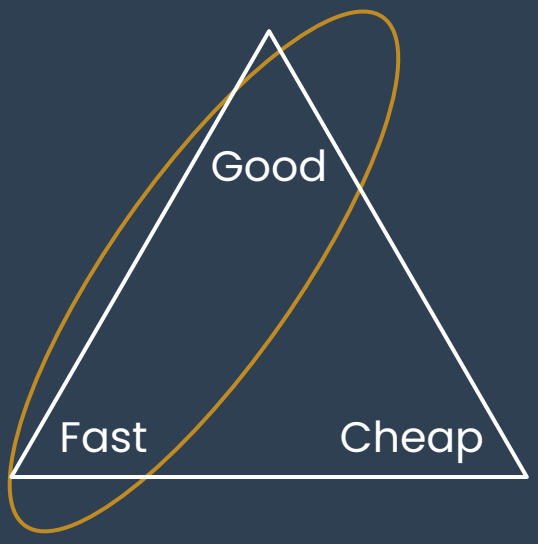
Those last 3 are what had happened with my client's rescue mission. A group of contractors had built the projects when they were greenfield and a couple early-career devs inherited them and did their best. Then the startup went in a new direction when they gained a better understanding of their market and the entire organization was still figuring out what that meant.



It's also helpful to understand the main stakeholders' priorities when trying to figure out what decisions led to where we are today. The project management triangle is a common heuristic for understanding the constraints for almost all types of projects, not just software, and you've probably heard it before, "Good, fast, cheap–pick 2."

If a team or organization consistently prioritizes the same 2 constraints, you're going to start having problems in the gap left by the third constraint.

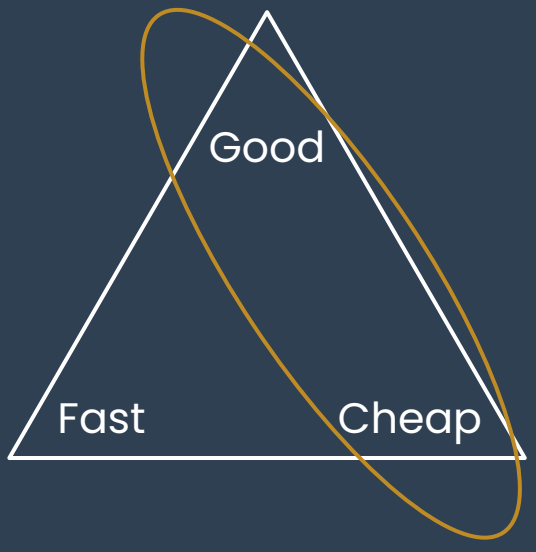
Project management triangle



@mercedescodes | mercedesbernard.com

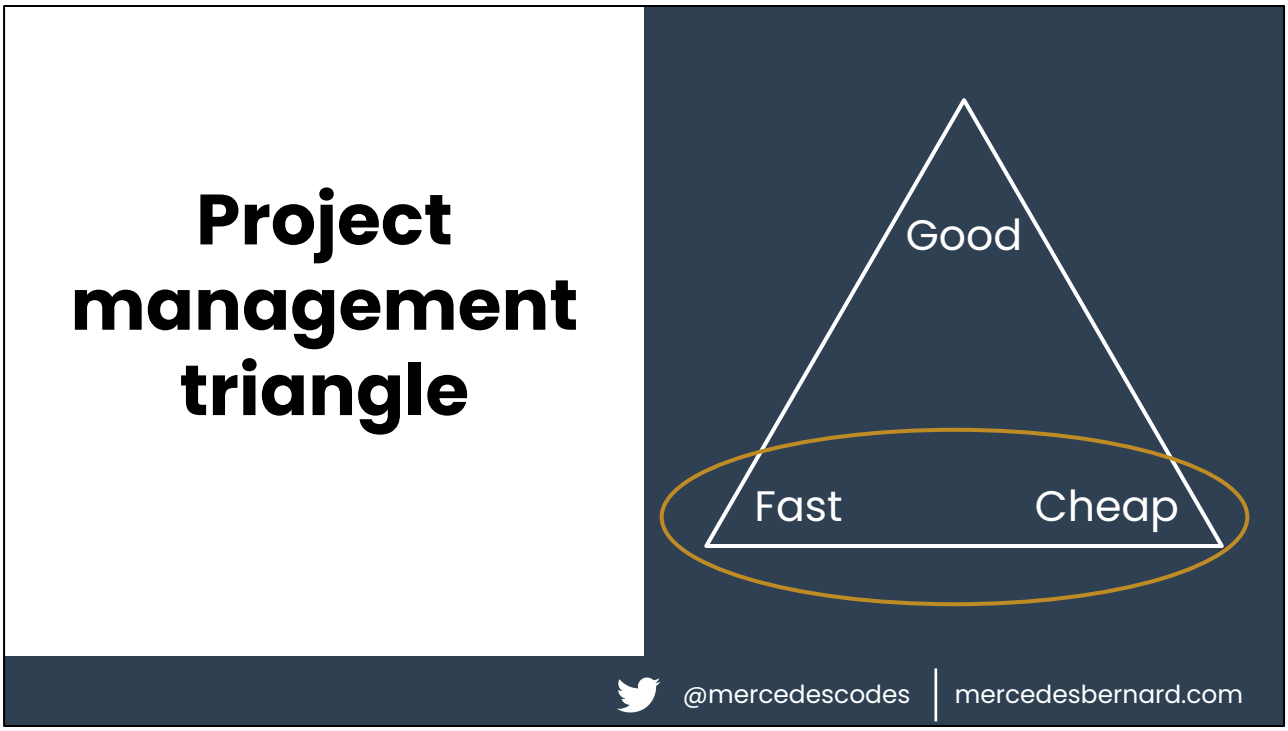
If you always choose good and fast, you could quickly run out of money.

Project management triangle



@mercedescodes | mercedesbernard.com

If you always choose good and cheap, you may never get to market in time to capitalize on your idea.



And if you always choose fast and cheap, you may have some extreme quality issues.

This is why it's ok to incur some technical debt on projects. Sometimes, the best decision for the business is to choose fast and cheap in order to meet timeline or budget constraints. And deciding how to balance project management constraints is a big part of managing and paying down technical debt.

However, for my recent rescue mission, it was clear that the organization did not balance project constraints. They always chose to prioritize cost and timeline at the expense of their product quality. This resulted in a "kick the can down the road" strategy until the projects got to the point where they had to be rescued.

Audit the existing state of the system

mercedesbernard.com

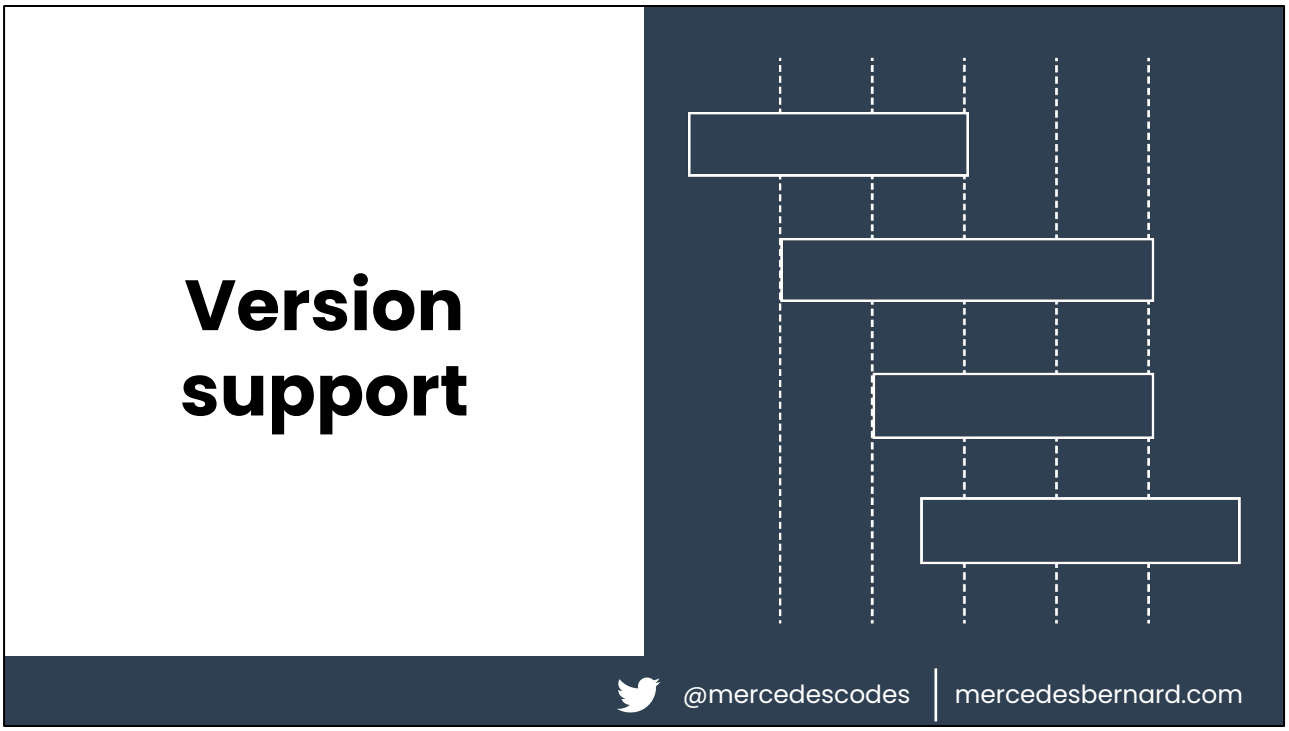
Taking the time to understand the business and the decisions behind the current state of the application allows you to remember the humans who built it and is key to the “empathetic” part in “gain an empathetic understanding.” But now we’re going to go over how to get the “understanding” part.

When you inherit code that you didn’t write, especially entire applications or systems (not just features), it can be daunting and hard to know where to start. And when you not only have to understand the code but also identify its weak spots and figure out how to stabilize it, it can be downright overwhelming.

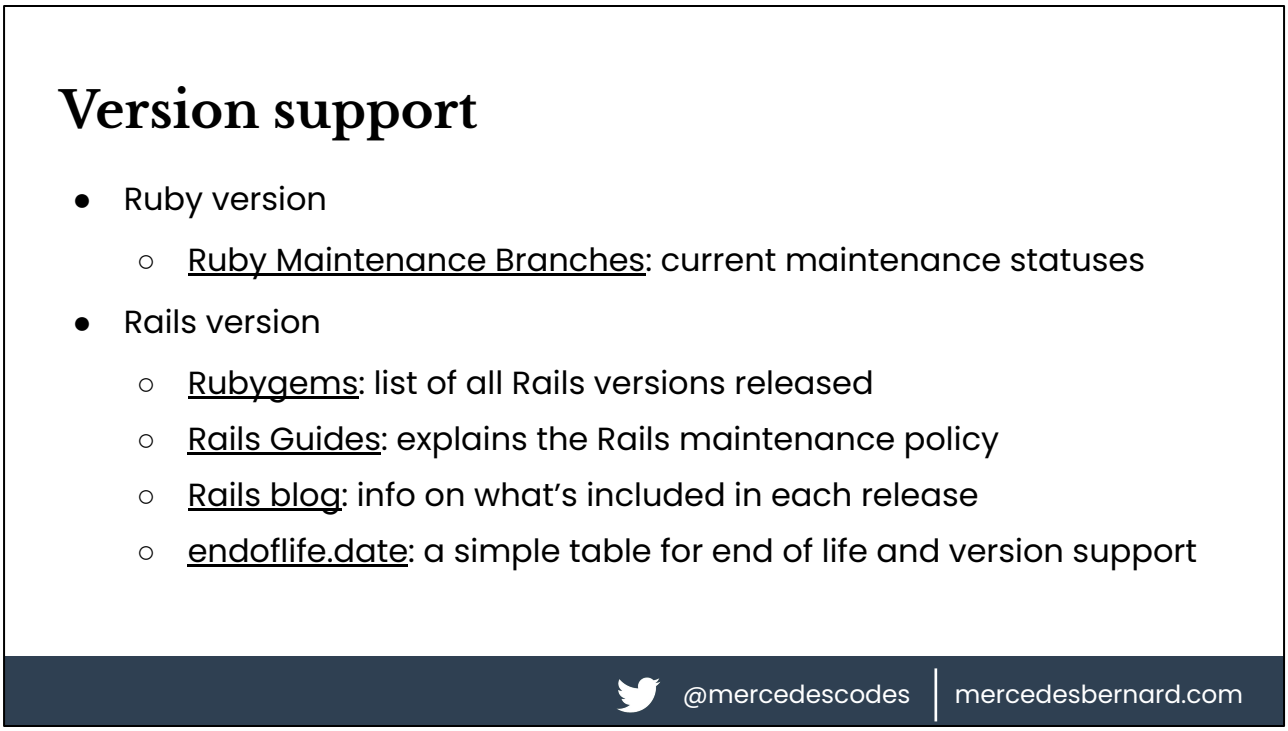
When I know I have a rescue project in front of me, these are the steps I take to understand the scope of the mission. Full disclosure, this can be time-consuming.

First, try to get the project up and running locally. In a best case scenario, the project is containerized and this is a 5-10 minute process. On a rescue project, I've never encountered a best case scenario.

Hopefully there is some documentation to get started. If not, you'll have to rely on industry standard practices for the language and framework you're working in. On a Rails project, you know you're going to start with `bundle install`. This isn't always enough but if you aren't able to get the project running in 30 minutes without documentation, that's a valuable data point that the issues in this codebase may be bigger than what anyone expected because there are unknown unknowns.



Then look up and document for yourself the language and framework version that this project is running on



For a Rails project, what version of Ruby is the project running? Check both your `.ruby-version` file and your Gemfile. What is the maintenance status of that version? Is it supported? Is it going to reach end of life soon? Did its EOL date already pass? The Ruby Maintenance Branches documentation will tell you the current maintenance status of the various Ruby branches.

Then check those same things for Rails. Look in your Gemfile (or Gemfile.lock if the project isn't strictly enforcing a Rails version) and see what version you're running and check the maintenance status for that version.

Rubygems has a list of all Rails versions released, Rails Guides explains the Rails maintenance policy, the Rails blog includes info on what's in each release, and endoflife.date is a great little website that provides a simple table to track end of life and version support.

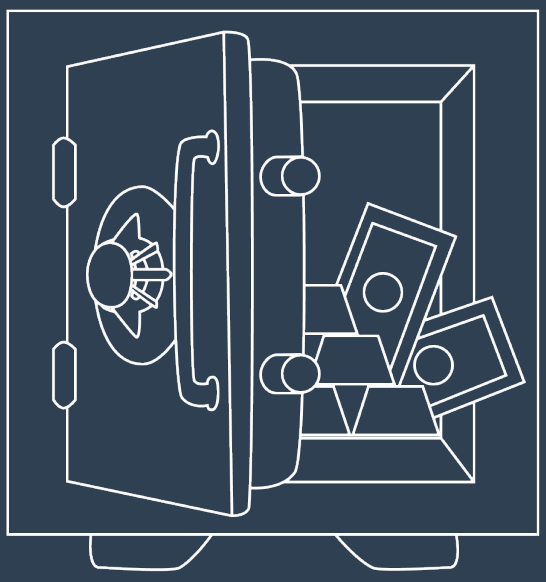
Version support

- Postgres
- Gems
  - `bundle outdated` to find outdated gems

And if there are other really important packages and dependencies, take a look at those. For example, what Postgres version is the app running on?

You can also use `bundle outdated` to find gems that have updates available. You won't necessarily have to upgrade all the gems but it can give you a good sense of where the project is at. You should pay special attention to the critical dependencies when deciding what to upgrade.

When we inherited our rescue mission, the Rails version had reached EOL 2 years prior and the Ruby version had reached EOL a few months prior. So we knew that we'd definitely have to upgrade those if we decided to stabilize these applications.



Security vulnerabilities

@mercedescodes | mercedesbernard.com

If your language and/or framework have reached EOL, they're not receiving security patches anymore. But even if they aren't EOL, there's still a high chance of lurking vulnerabilities if things haven't been kept up to date. You'll want to make sure you do a security audit so you know what you're working with and how much work you'll need to do to get things secure.

Security vulnerabilities

- Static analysis
  - Brakeman
- Check your certs
- Data encryption
- Authenticaton and authorization
- Sensitive credentials
- Helpful Rails security audit list

Security is an entire discipline unto itself and may not be your bread and butter. We can't specialize in all the things. So if you need to do a security audit and security is not your specialization, here are some tools and tips I like to use on a Rails project to make that easier.

- Brakeman - A static analysis security vulnerability scanner for Ruby on Rails applications
- Check your certs! Is the web traffic over TLS? Is there an SSL cert in place?
- Is data that should be encrypted (i.e. passwords) encrypted?
- Is there authentication and authorization for resources that should be private for users?
- Where are sensitive credentials for the system stored? Make sure they aren't in the source code or checked into source control anywhere

There's probably a lot more things to be aware of, I'm not a security specialist either. But with static analysis and some general web security know-how, you can get a sense for how vulnerable the application is and how much time you may need to dedicate to remediating security issues. I've included a Rails security audit list as well to help.

Data

After getting a general lay of the land, we should look at the data.

Data model

- Does it make sense conceptually?
- Referential integrity
  - Are the associations in your models enforced through foreign keys in the database?
  - Are the foreign keys required unless the association is optional?
- Normalization
- Model associations

When I'm talking about looking at the data, I mostly mean the data model. In a Rails app, we're typically dealing with relational databases so that's what I'm going to focus on.

The first thing I think is important to look at in the data model is also the most general–can you understand it? And I mean this on the most basic level, does it make sense conceptually or does it leave you scratching your head?

This is a good gut check to get a sense for how challenging and time-consuming it's going to be to work with as you dig into the other parts of the code.

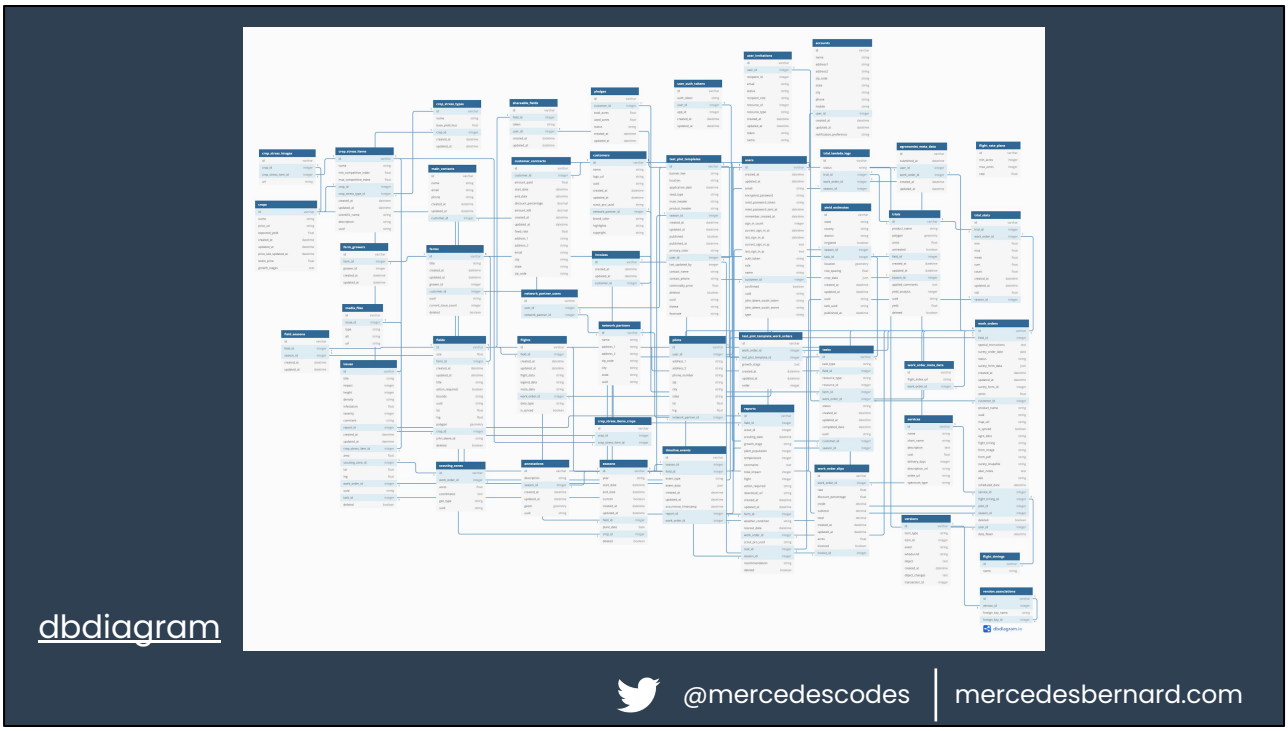
Once you have a feeling about how understandable the data model is, take a look at referential integrity. Are the associations in your models enforced through foreign keys in the database? Are the foreign keys required unless the association is optional? I've seen Rails projects in the past that rely exclusively on model associations and don't add foreign keys to enforce them.

If you have strong referential integrity, meaning the answer to both of those questions was yes, you can have higher trust in the data that if you had to migrate it, you would be able to without a lot of headaches managing edge cases or error cases because the records that are referenced actually exist.

Last, take a look at how normalized the data is. Does the level of normalization make sense? In a relational database (which we're using in most Rails apps), we want a

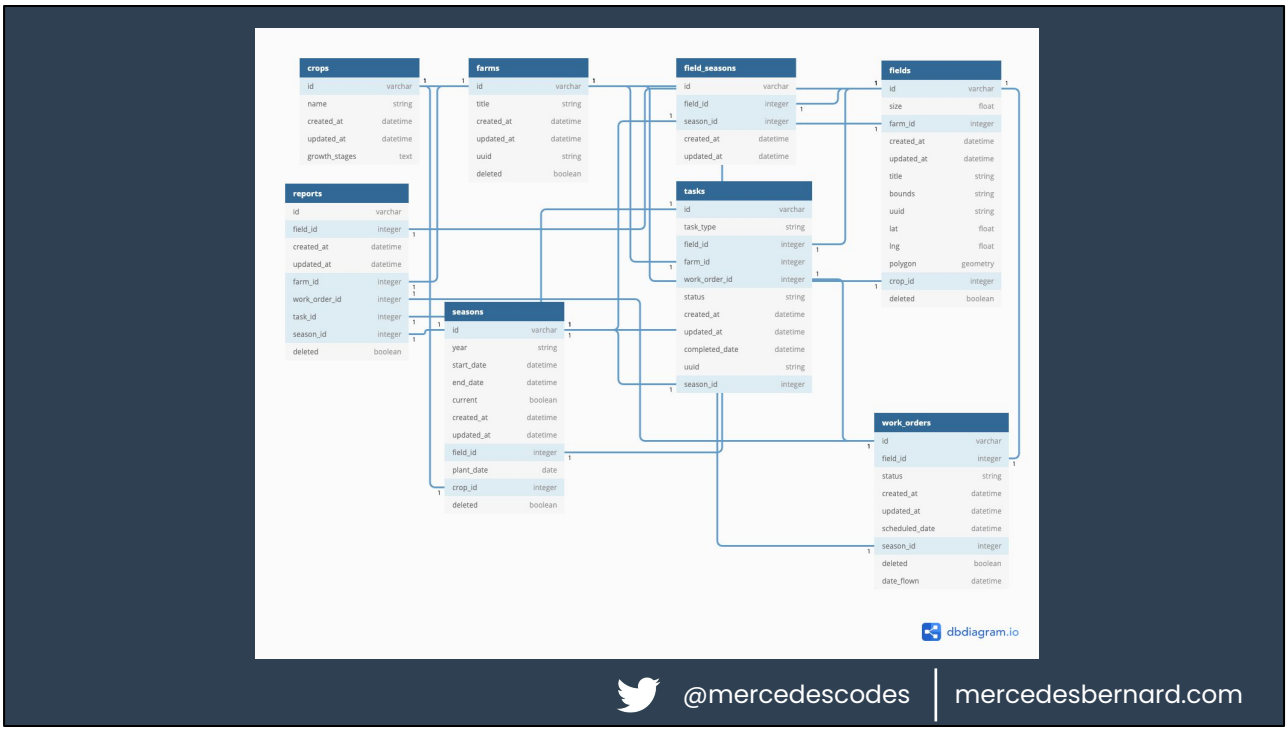
high level of normalization meaning we want to minimize data redundancy. Normalization is why we don't store everything in one table with data duplicated between records (think if we tried to flatten a has many relationship). It's also why we have has_many through associations rather than storing multiple foreign keys everywhere to get those direct relationships.

Sometimes there may be performance reasons to intentionally denormalize some data but most of the time, we prioritize data integrity and a higher level of normalization means you can trust your data more.



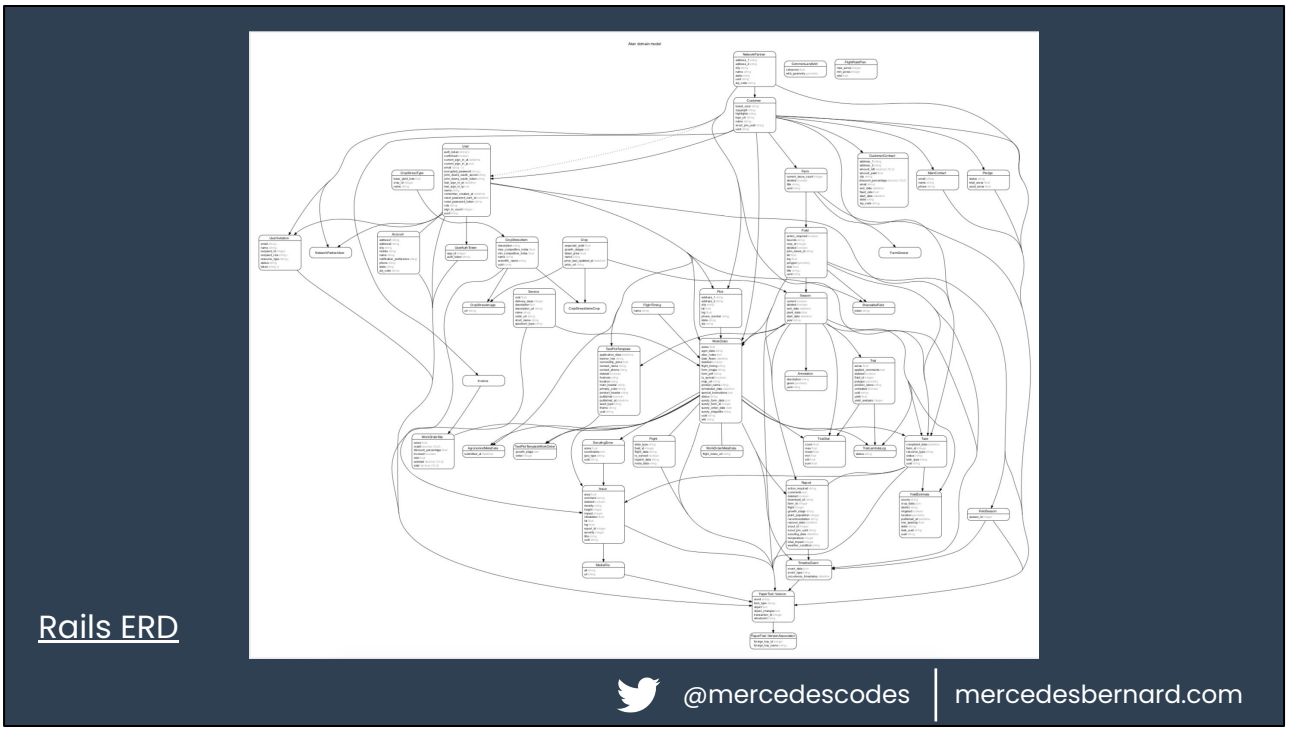
There are 2 tools I like to use to get a quick visual representation of the data model.

The first is dbdiagram which is a web app that you can upload the `schema.rb` file to. Because dbdiagram uses your database schema, the resulting visual will tell you if you have referential integrity in the database. Can you see the foreign key relationships between tables?



This diagram can also start to tell you about the level of normalization you have. If you see lots of foreign key relationships that seem to be duplicating a has-many association (like in this small subset of relationships), then your data is not normalized and you have some data redundancy.

For normalization, you also want to pay attention to duplicated columns across records that could be refactored into their own table but that requires more manual investigation than this diagram can give you.



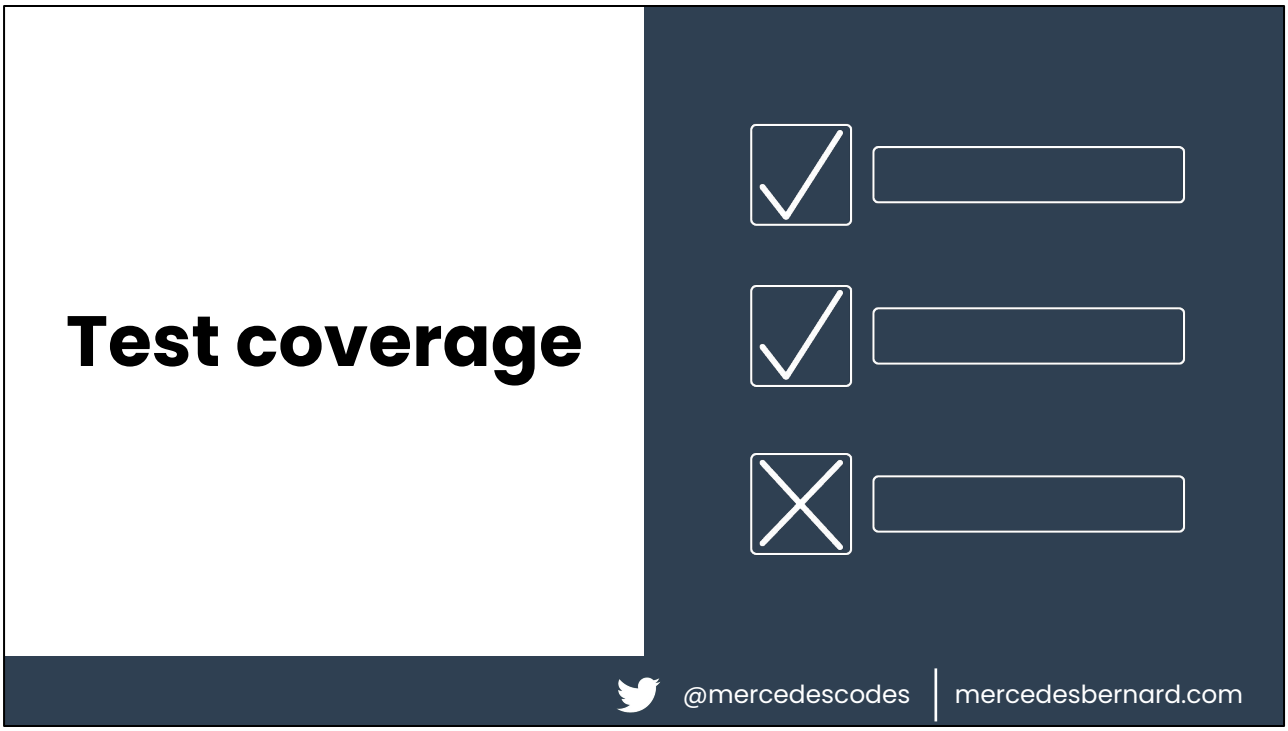
The other tool I like to use for evaluating a Rails data model is Rails ERD which is a gem that generates a diagram from the Rails model associations. It has a lot of customization options but it will show you a data model built from the associations in the application code. This is helpful for checking if the foreign key relationships in the database match the associations in the model. I often find that there are more relationships in the ERD generated from model associations than in the ERD generated from the schema which tells me that there are missing foreign key relationships in the database.

As you can see from the last few diagrams I have shown from our rescue mission project, I mostly just learned that the data model was a mess. There were so many unnecessary relationships throughout the database and the application code. Even though there was technically referential integrity, I couldn't tell if it matched the logic that was modeled in the application code. And there were things duplicated everywhere so I didn't have a lot of trust that foreign keys throughout the code were properly maintained in sync with the other relationships due to the lack of has_many through relationships (which are modeled as dotted lines in this ERD diagram).

Code

@mercedescodes | mercedesbernard.com

After digging through the data layer, we're going to finally look at the application code.



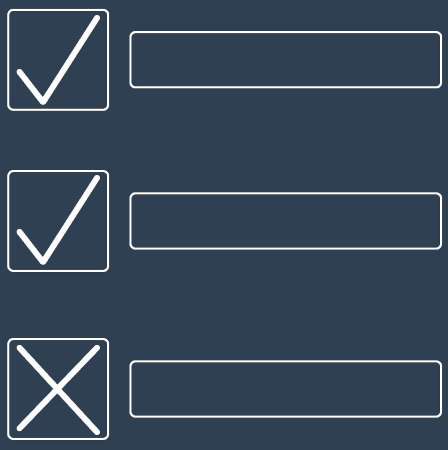
Start by looking at the test coverage within the application. Are there tests? Are they well-written? Do they test real things? For example, if you have a controller test, is it testing the shape of the returned response or just that it's a 200?

Auditing the test coverage in an application tells you the risk factor for changes/refactors. The less test coverage there is, the higher your risk that you'll introduce unintended regressions.

Good test coverage also serves as amazing developer documentation to tell you what a feature is intended to do and how to use it within the code.

Test coverage

simplecov

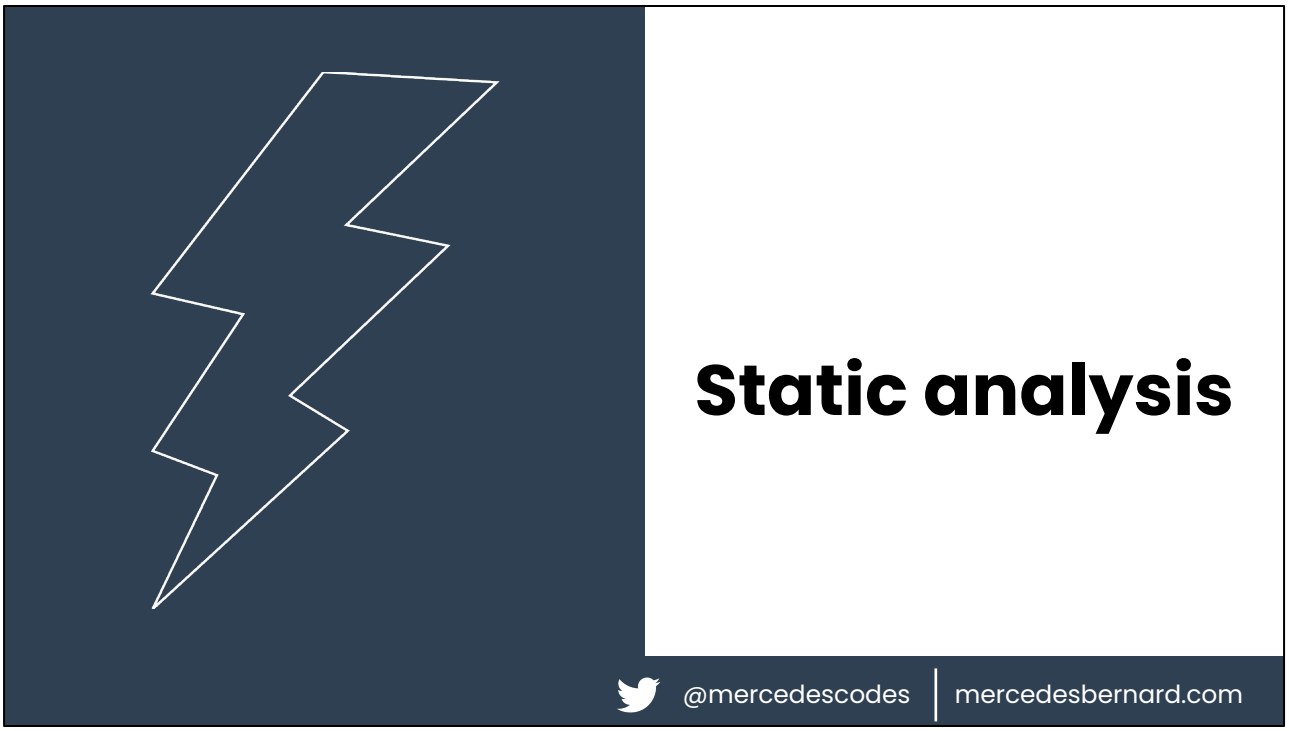


@mercedescodes | mercedesbernard.com

With a Rails project, you can use a tool like simple-cov to get a coverage percentage and identify paths in the code that are not tested.

On my rescue project, there were no tests. It was no wonder that the client was afraid to make a change because there was no feedback loop to catch if a change broke something in the code.

When there's no test coverage, you should budget time when stabilizing to add test coverage. In my opinion, this is non-negotiable. The alternative is far too expensive.



We've talked about static analysis during the security audit and a little bit when looking at test coverage but static analysis is super helpful for a dev unfamiliar with an entire codebase. Static analysis is when we examine code without running it usually trying to measure its quality in some way. And there are lots of tools to help us do that.

You can learn what parts of the code are complex and will be hard to understand, which parts change a lot (and if any of those are also the complicated parts), which parts are duplicated and more. When taken as a full picture, all of this can tell you how easy it will be to work with this code.

Static analysis

- Rubocop: static code analyzer and formatter
  - How idiomatic is the code?
- Reek: measure “code smells”
  - What areas of the code might be hard to maintain?
- Flog: measures assignments, branches, and calls
  - How hard is this code to test?

On a Rails projects, some tools you can use for this are

- Rubocop: How idiomatic is the code?
- Reek: What areas of the code might be hard to maintain?
- Flog: How hard is this code to test?
- Flay: Where is there code duplication?
- Turbulence: Where are the good candidates for refactoring?
- RubyCritic if you want a single report to wrap up some of the gems above

When you find areas of the code that keep popping up during your analysis, high complexity, high churn, lots of duplication, you can add that to your list as priorities for refactoring once the project is stable.

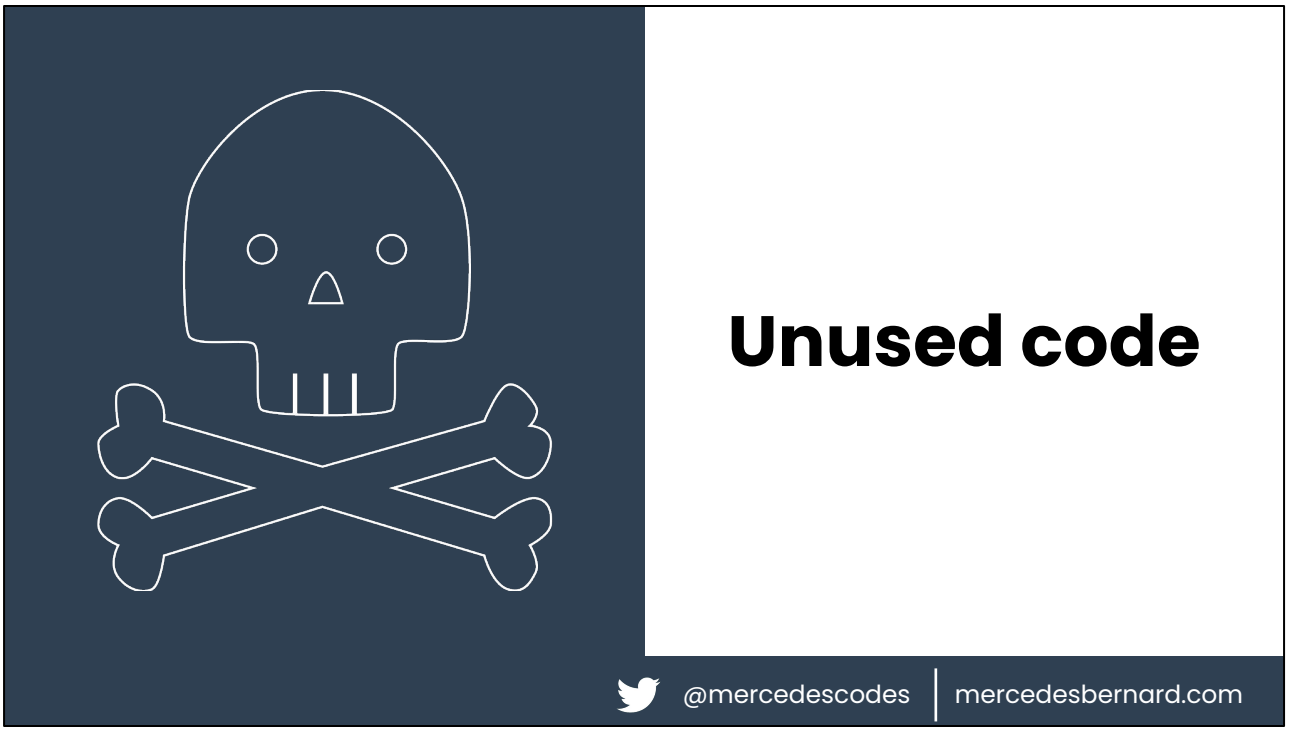
Static analysis

- Flay: finds areas of the code that are similar, i.e. not DRY
  - What parts of the code are similar to each other?
- Turbulence: evaluate code complexity and churn
  - Where are the high churn, high complexity parts of the code?
- RubyCritic: Wrapper around Reek, Flog, and Flay

On a Rails projects, some tools you can use for this are

- Rubocop: How idiomatic is the code?
- Reek: What areas of the code might be hard to maintain?
- Flog: How hard is this code to test?
- Flay: Where is there code duplication?
- Turbulence: Where are the good candidates for refactoring?
- RubyCritic if you want a single report to wrap up some of the gems above

When you find areas of the code that keep popping up during your analysis, high complexity, high churn, lots of duplication, you can add that to your list as priorities for refactoring once the project is stable.



Another metric to look at in a code base is to try to find unused code. I have a lot of clients who don't want to delete deprecated code because they like to have a record of it despite having source control. But keeping unused code around is an expensive maintenance cost. As the codebase evolves, your unused code may break tests (if you have any) as database constraints or associations change. And that code may confuse developers who think that they need to refactor and keep this code in a ready-state despite being deprecated. Unused code can give you a lot of false negatives during your audit and future stabilization.

If you find unused code, add that to your list to remove. We don't want to invest in it any longer.

Developer Experience

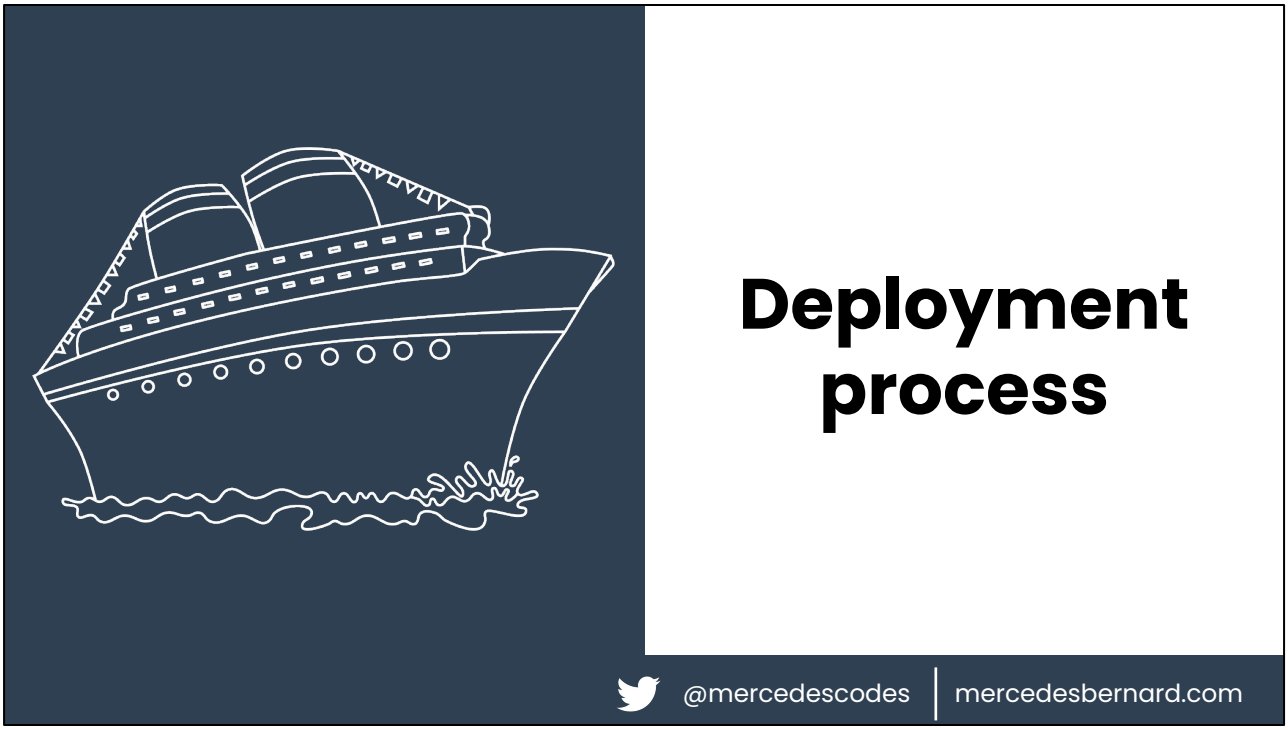
@mercedescodes | mercedesbernard.com

Finally, the last couple pieces to audit are related to developer experience. Happier devs are more productive devs and when we have a code base we feel comfortable working with, we tend to be happier :)

First up, is there error monitoring within the system? If something goes wrong in one of your deployed environments, how would you know?

When you're thinking about stabilizing an application that is in production and generating revenue, causing issues or outages for users during incremental updates is a big problem. And rescue mission projects aren't usually very resilient. So you'll want to know as soon as something goes wrong so you have as much time as possible to roll back or fix it.

If there is no error monitoring, make sure that gets added to your list as a high priority item to resolve.



Speaking of resilience, how easy is it to deploy the application? How easy is it to roll back to the previous state in the event of regression?

Most projects these days have some sort of CI/CD pipeline but that's not always the case and it's good to understand how changes make their way to production. What is the deployment strategy? What is the branching strategy? Is this all documented? And maybe even most basic of all, can this application be deployed?

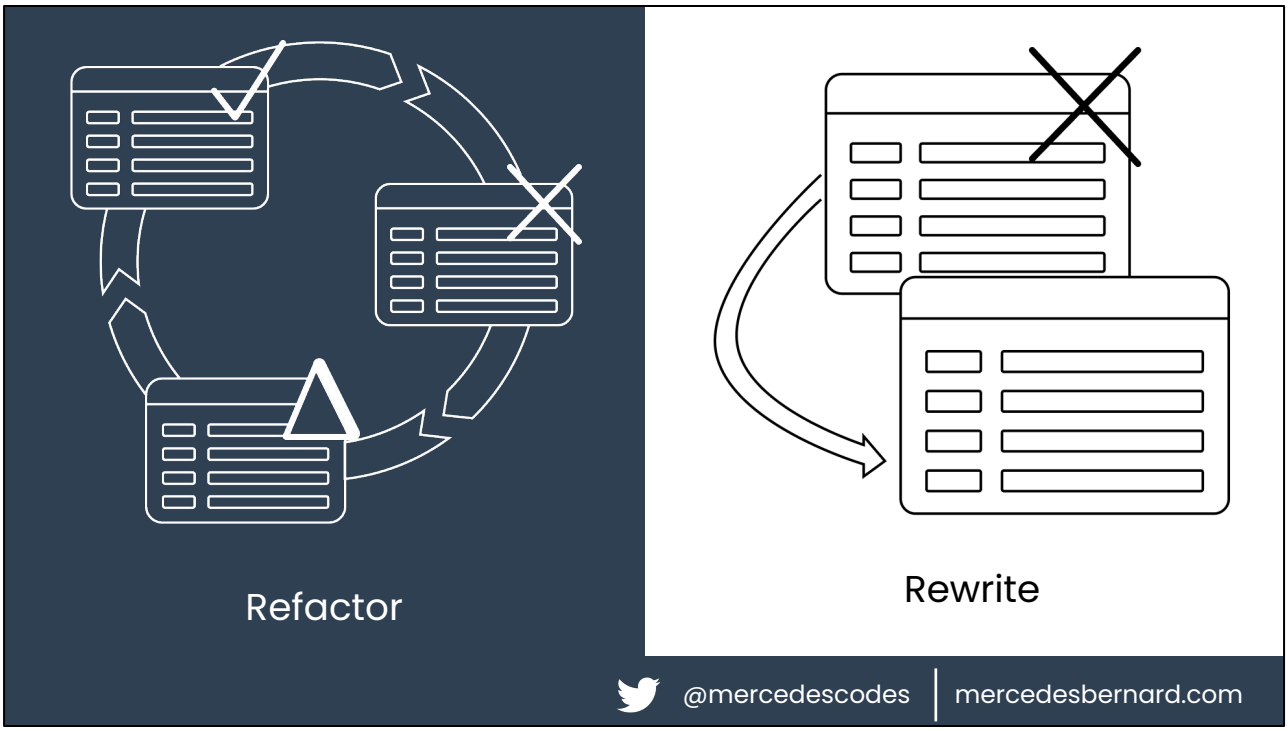
Hopefully all those answers positive and your life will be easy. But if not, then you have something to prioritize in your stabilization work.

Edge case requirements

@mercedescodes | mercedesbernard.com

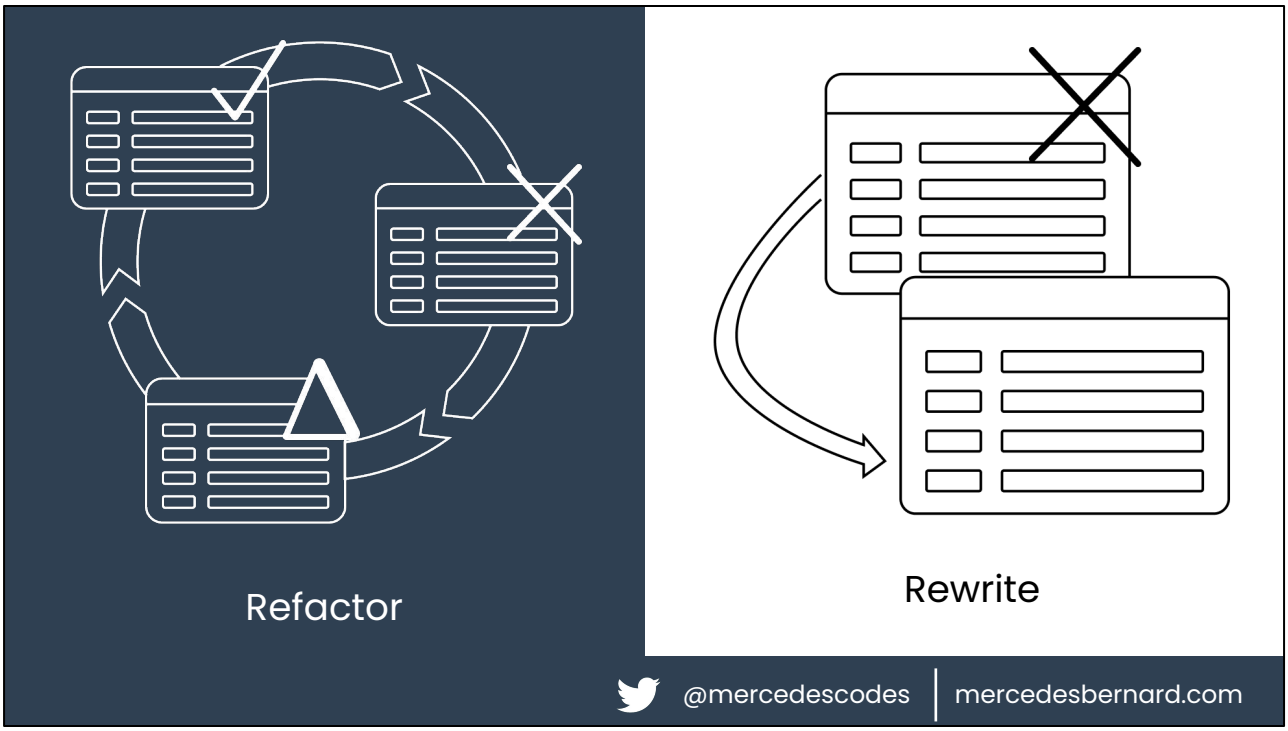
Finally, get a sense for edge cases that you should be aware of before you embark on the project. One that has burned me in the past on a different project was that we had to support an older version of Internet Explorer and didn't realize it until we were pretty far into upgrading frontend packages and finding all the things we broke was a challenge, especially the styling.

Whew, that was a lot. But now that you have all of this information, you're able to make a plan and decide on what are the best next steps.

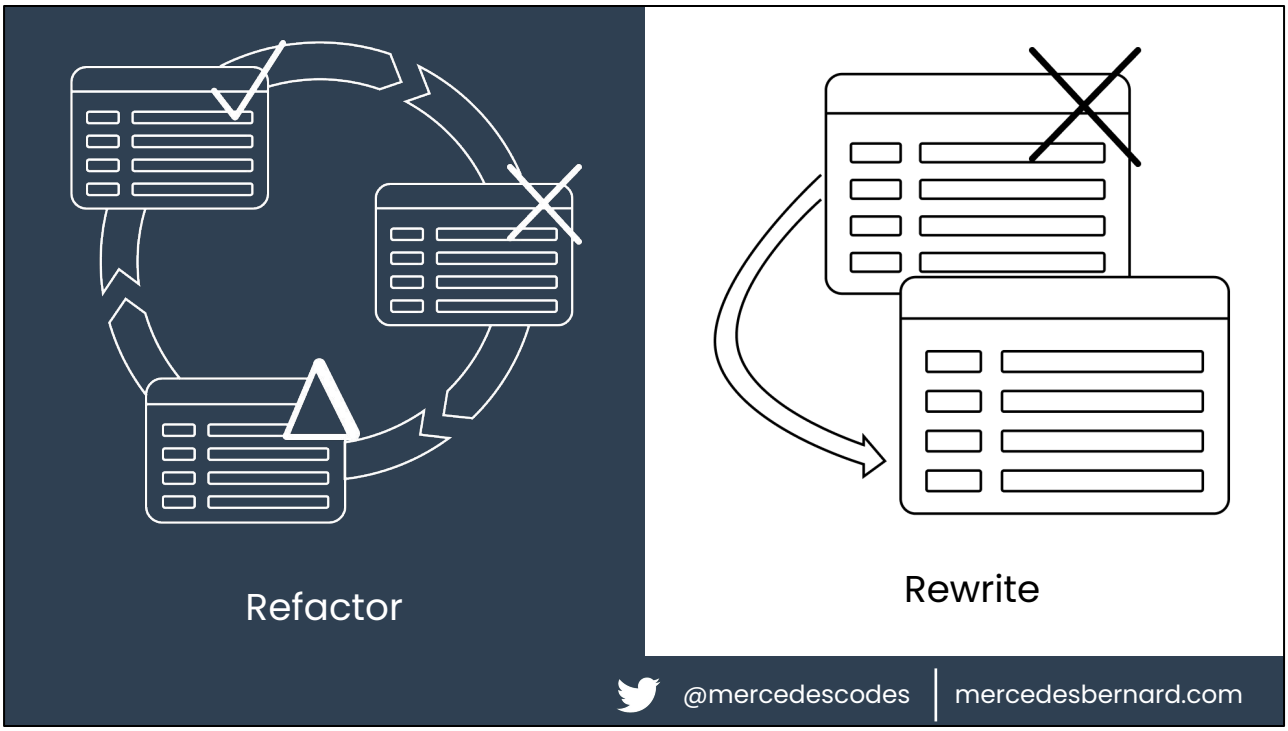


The first thing you have to decide is whether or not the application is salvageable. And let me start by saying 9 times out of 10, it is.

I've seen a lot of teams and clients choose to rewrite an entire application over refactoring. And I think this happens for 2 main reasons. One being that I don't think teams are given the time to invest in understanding the current state of an application when making this decision. As I was going through all the different things to look at, if you were thinking "this is too much" or "I'd never have time to dig into all this" then you might be on this kind of team. With all of the unknowns and ambiguity around what's in the application, where the problems are, and where the stable bits are, it's hard to feel like there's any incremental path forward. It feels much more manageable to write code from scratch rather than make incremental improvements to someone else's code. Writing your own code feels safer.

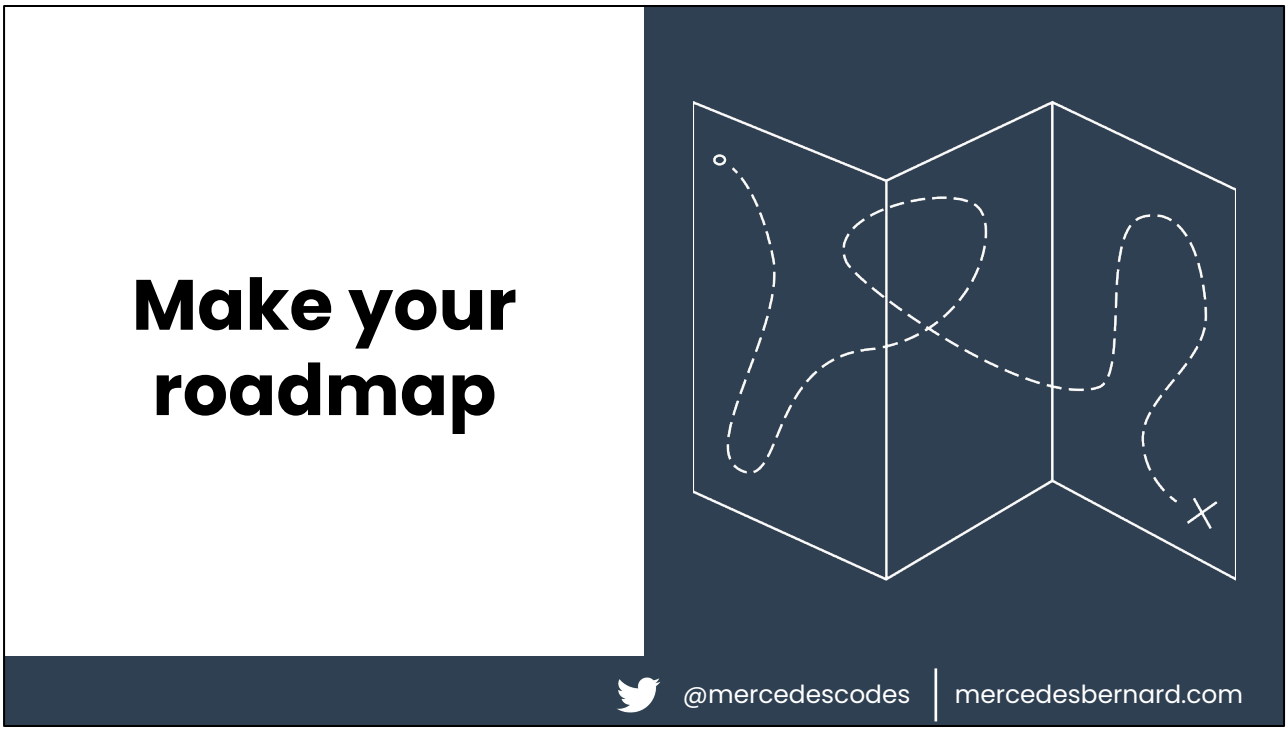


The other main reason I've seen teams prefer rewrites over refactors is that as an industry we are serial under-estimators. We've all had those moments of, it will only take a day and then 4 days later, we're still working on the same thing. Rewriting an application from scratch always (ALWAYS) takes longer than anyone thinks. There are features we forgot existed, the data migration is more tedious and trickier than we thought, there is new tech or patterns we play with that we decide we don't like and we end up refactoring the new thing as we go, and styling takes longer than any dev predicts (we have to stop giving CSS the short end of the stick).



Now, I know I'm saying all of this but without research studies or hard numbers to back it up. There's a little bit of trust in this talk too. Based on my experience as a consultant for over 9 years, I've never seen a rewrite project land without going significantly over time and over budget. And I've only ever seen one repository where I whole-heartedly recommended rewriting instead of refactoring.

Of the 4 repositories my client brought me to save, we refactored and stabilized 3 of them. The fourth repo was not making them any revenue, did not have active users, no longer met their business needs after they had a better understanding of the market, and was built on a hand-rolled React Native library that hadn't been maintained in years and was in a state of deadlock due to strictly enforced package versions meaning that we couldn't upgrade anything without also upgrading that package which was impossible. Given that there were no users and no business case, it didn't make sense to invest in this app at all and we recommended that they don't rewrite it but instead take time to research what value they could get from a mobile app and then write *that* app.

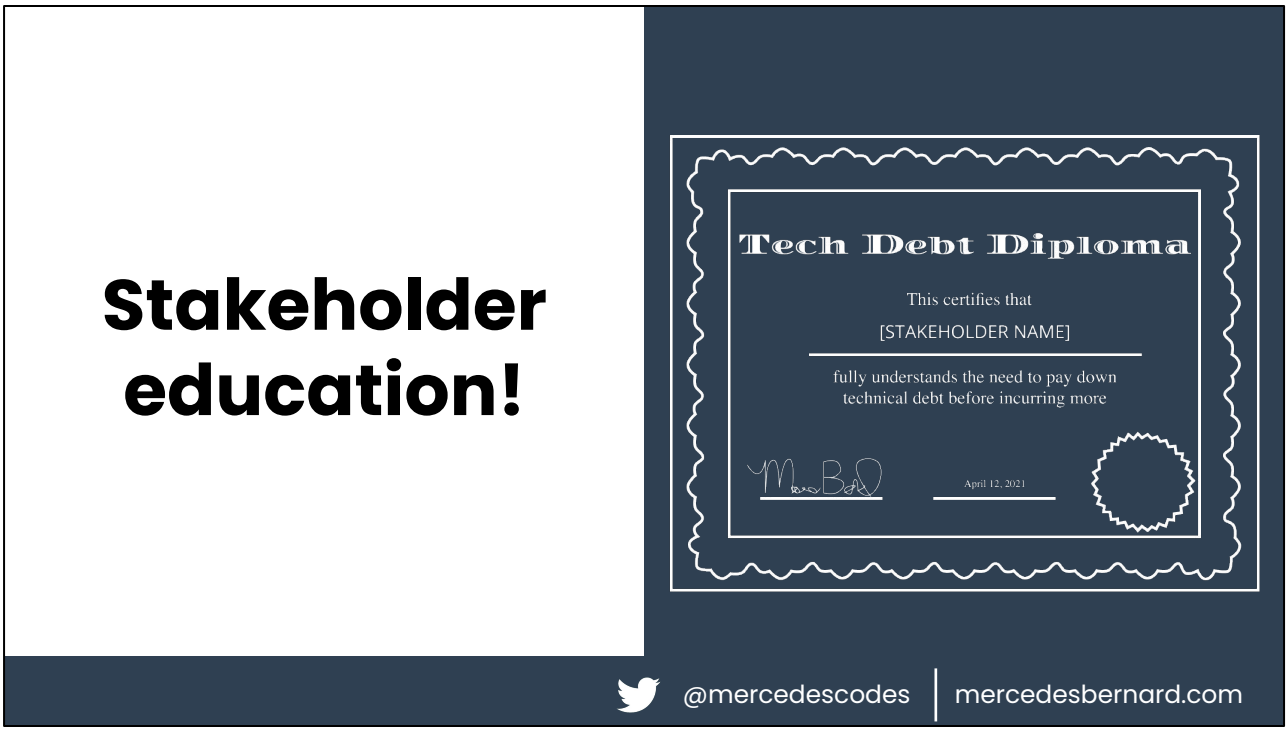


If you went through all the audit steps before and have a list of things to address–automate deploys, upgrade the language version, patch security vulnerabilities, add error monitoring, clean up the data model–you have the start of your roadmap.

It's time to take that list and put them in priority order. Your highest priority items are going to be those which will enable the team to ship working code easily. It's impossible to make incremental improvements to an application if there's no easy way to get code out the door. That friction will result in more things being batched in every release which increases the surface area for regressions and will continue to erode trust in the application.

If the project doesn't have automated deployment workflow, that should be really high on your list if not the most important.

After you have your prioritized roadmap of the work needed to stabilize the application, use it to make an estimate. After the audit, the team should have an understanding of what needs to be done and you'll have it broken down in your list. Go through and create realistic estimates including time for manual regression testing. Your estimate will be high. This isn't going to be a sprint of stabilization. That's why it's a rescue project. And that's ok.



Once you have the work identified, prioritized, and estimated, it's time for stakeholder education. It's really important not to skip this step. In order for the team to have the time they need to stabilize the application and to ensure success, all stakeholders need to understand the stabilization work.

First, it's important to explain why the issues are a problem. Non-technical folks may not understand why investing time in creating a stable CI/CD pipeline is important and may want to deprioritize it because "we've gotten code to production before." Explain how automated tests and deploys mean that the developers can ship code more quickly with these tools.

If you have to spend a lot of time re-working the data model, explain the risks of data migration when there isn't data integrity or long-term consequences of denormalized data.

When you're explaining the parts of the roadmap and their purposes, try to keep in mind what your audience's values are. If they care about time and budget, frame the purpose and need for the work to show how it will maximize future budget. If they care about the customer experience and are concerned about not getting new features, frame your argument to show how increased stability and lack of regressions will increase customer trust and lead to a better experience.

When you're stabilizing a rescue project, you may not be able to add new features. Actually, let me rephrase, you shouldn't add new features while stabilizing a project. So the last step of stakeholder education is clearly setting expectations about when they can get new features they may want. And as the project starts to get better and better, they'll become more excited and may not want to wait for you to finish your work.

Prepare a milestone in the roadmap for when the project will be in a good place and the team can start balancing further stabilization work with new features. Share that milestone with your stakeholders. Then when they start asking for things, you have something concrete to point to that was agreed upon up front.

For my rescue project, my stakeholders obviously cared a lot about budget. They wanted everything as cheap as possible. When advocating for this refactor of their projects rather than rewrites, I explained how they would be saving money. When they first saw the stabilization estimate, they thought they could get an entirely new application for the same price, but we had some frank conversations about how their tendency to sacrifice quality got them into this state so it was time to invest in well-written code to get them stable software they could continue to build on.

That "continue to build on" part was key because besides budget, adding new features was their other highest priority. They had a list of things they wanted so they could market their product to investors and raise another round of funding. When I was explaining the benefits of stabilizing their projects, I explained how a high-quality

foundation would also allow devs to build features quickly because they didn’t have to wade through the cruft; there would be shorter feedback loops. They were a little skeptical but quick delivery was a huge selling point for them.

Once you have a plan for stabilization and you have stakeholder buy-in, it’s time to get to work.

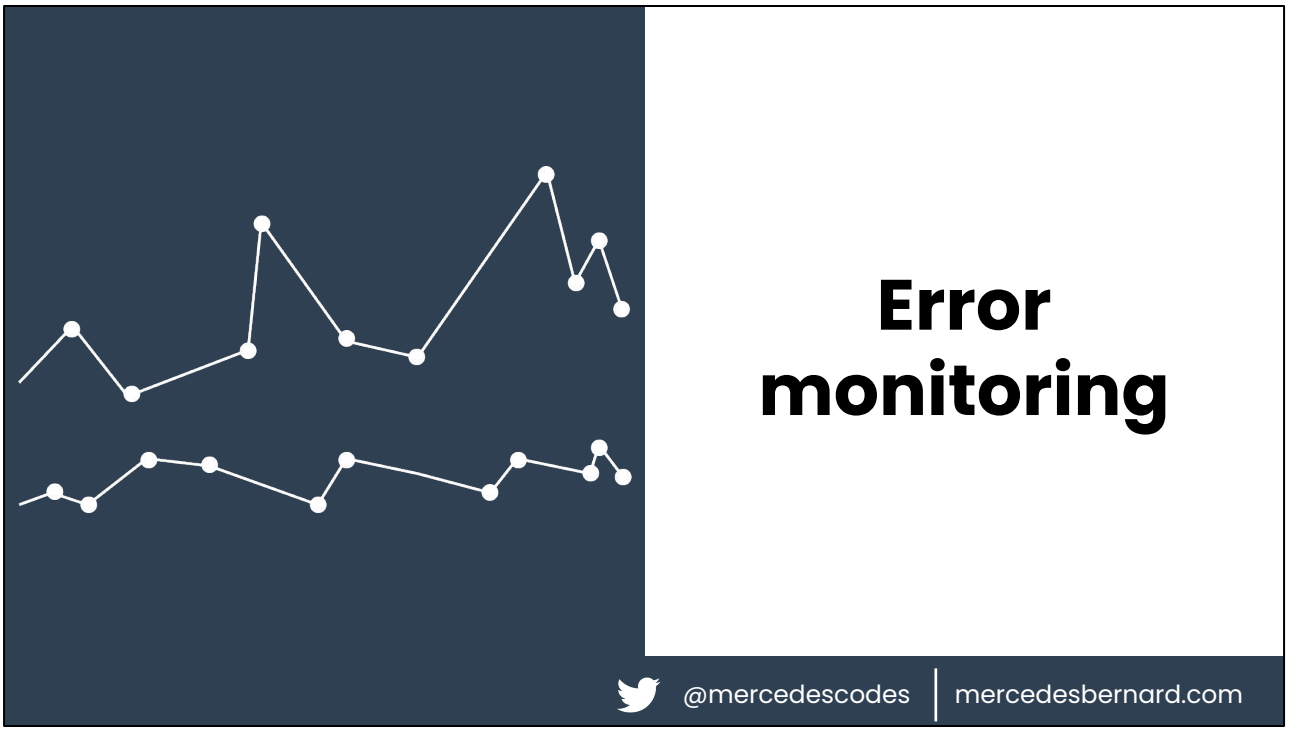
Possible roadmap

1. Working dev environment
2. Automated CI/CD
3. Error monitoring
4. Security vulnerabilities
5. Test coverage
6. Upgrades
7. Incremental refactors

For our rescue project, our prioritized roadmap looked something like this. Yours may look different! Depending on what is important to you and your team as well as the current state of the project, you may have upgrades higher on the list or you may not have automated CI/CD on yours if your project already has a working CI pipeline.

For us, the project was being deployed via Heroku and even though the Ruby and Rails versions had reached EOL, there was still a viable Heroku stack for them so we worried about other items before upgrading so that we'd have test coverage to catch if the upgrades broke anything.

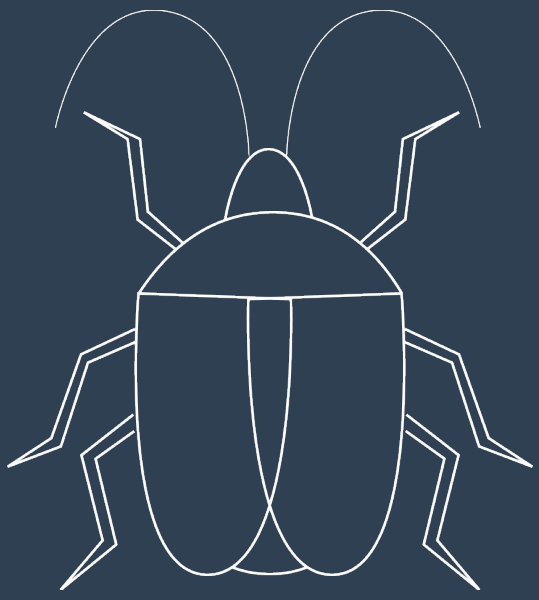
It’s going to be challenging to stick to the plan you made as you get your fingers in the code. Your team may see code they want to clean up or glaring bugs they want to fix, but it’s important to be methodical and organized as you stabilize the project. You put a lot of thought into the best way to stabilize the application, so it’s important that you don’t try to spin too many plates at once.



Remember when we were talking before about how rescue projects tend not to be very resilient? So we need to find easy, low-friction strategies to add resilience early in the stabilization process.

Adding error monitoring is one of those low-hanging fruits. It won't make it easier to resolve a problem, but it will quickly give you feedback if a regression or outage occurs. If your project doesn't have error monitoring, make sure you add this early.

Test coverage

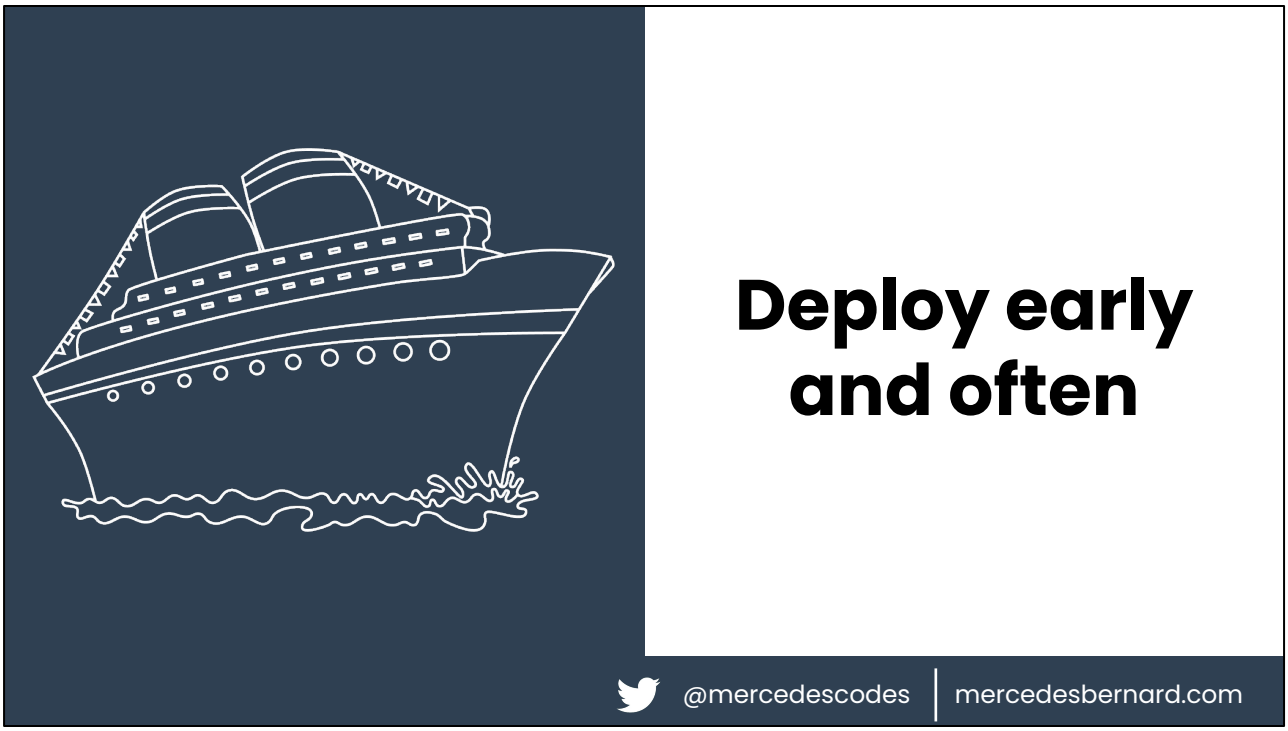


@mercedescodes | mercedesbernard.com

When you get to the point that you are adding test coverage, it may feel counterintuitive but test the bugs. We're not ready yet to make intentional behavior changes because we don't have strategies in place to catch regressions. Even though it feels bad to leave broken code, make sure you test the bugs. You can add info to the Rspec context block that this is bug behavior and should be fixed.

When you're adding tests, make sure you agree as a team what your desired coverage is. 100% coverage would be ideal to give you confidence in catching unintended behavior changes, especially if you are not the team/developer that wrote the code, but this isn't necessarily realistic. Pick a percentage and strive for that in all the files you test.

Timebox your time spent on this task and focus on the critical path (user flows, revenue generating flows, etc). If you have features that are almost never used, they are less important for you to test and if you run out of time, it's ok.



As you resolve security vulnerabilities and complete your upgrades, be sure that you are deploying those changes early and often. Smaller deploys are better. If you deploy 3 changes a day, that's better than 1 deploy a week with 15 changes. When a deploy is small, you can have a lot of confidence that you know what caused a regression if you find one. And rolling back small deploys are often easier because there's less side effects in small deploys. That's the whole selling point for CI/CD pipelines, right? You don't need me to tell you how great lots of small deploys are :)

If you are at the point in your stabilization plan where you are refactoring and making database changes, you often need multiple deploys to get those changes to production anyway.

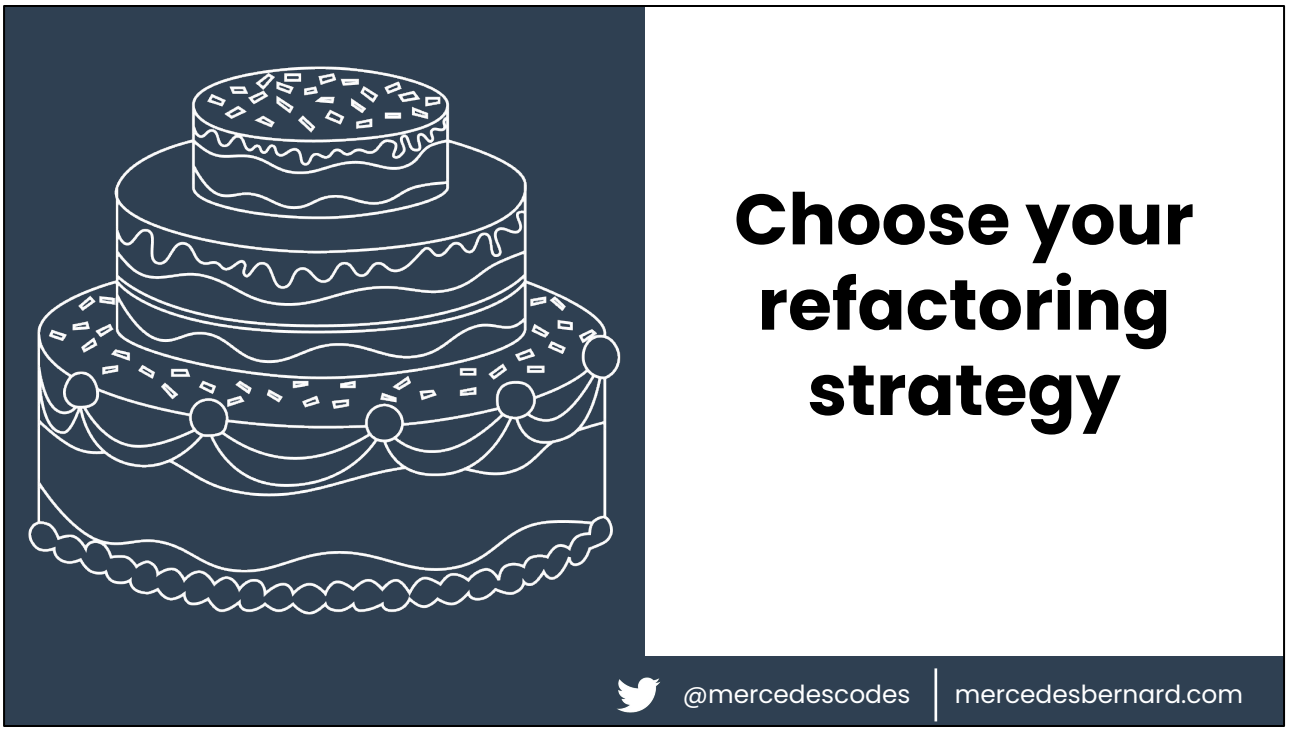
For example, if you want to move data from a deprecated table or column to a more appropriate place, you'll usually do one deploy to add the new table/column and a rake task or migration that copies the data from the old location to the new location. Then you can QA and make sure that the application code using the new location looks good. Once it does, you do a second deploy to remove the old table or column. And QA once more.

This is why having an easy to use deployment workflow is so important.

Incremental refactors

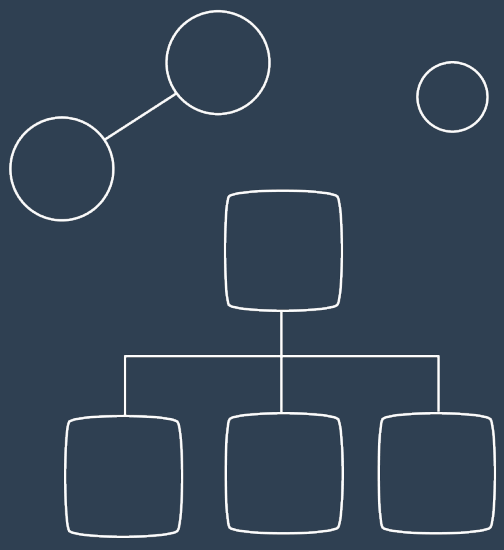
@mercedescodes | mercedesbernard.com

Related to small deploys are incremental refactors. We can only do small deploys if we make small changes. Now that we should have some test coverage, we should try to make small, isolated changes as much as possible.



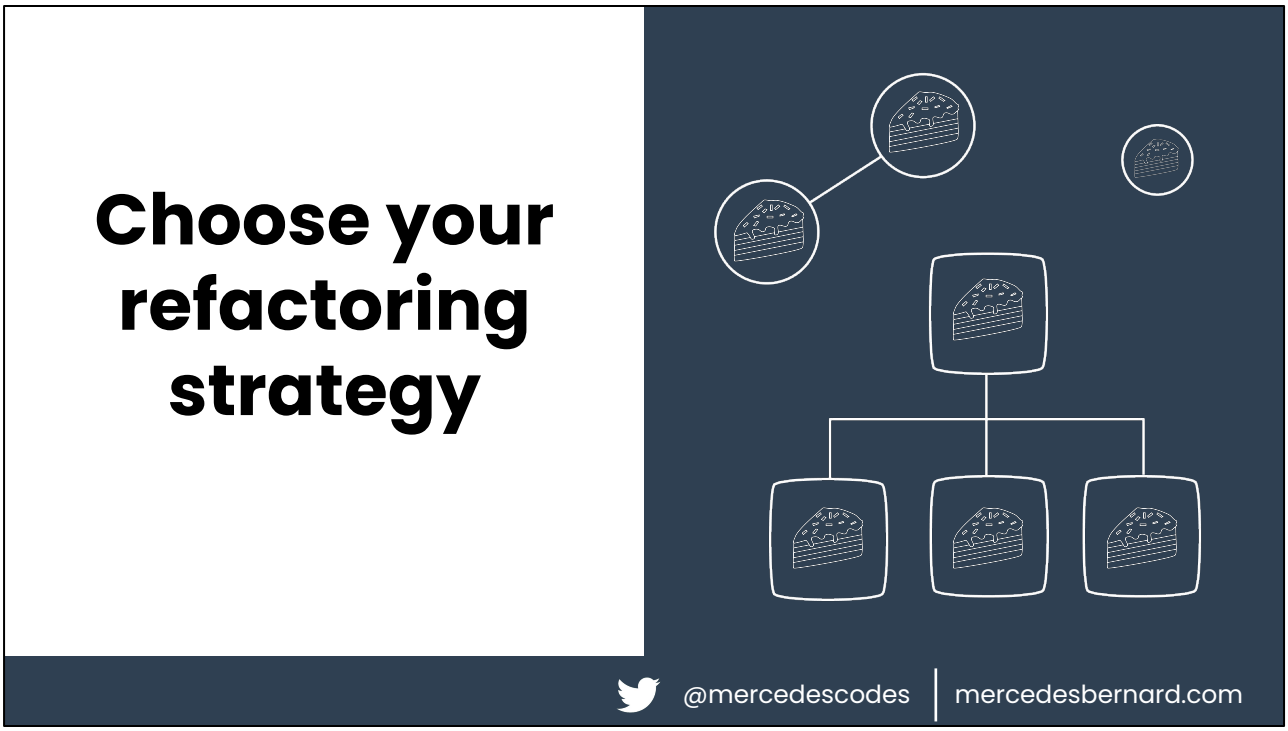
And when refactoring, you have to choose your approach. Do you want to refactor layer by layer? Focusing on the data layer, then the application later, and the UI last?

Choose your refactoring strategy



@mercedescodes | mercedesbernard.com

Or do you want to go domain by domain within the code instead? Maybe focus on user management then search then reporting, etc.



On our project, we did a combo of both. We had a super complicated, denormalized data model (remember those diagrams from earlier??) so we would pick one domain, start at the database and refactor layer by layer from db through business logic to presentation layer with many many small deploys until that domain was stable. Then we'd pick the next domain and do the same thing.

Lessons learned

@mercedescodes | mercedesbernard.com

So what lessons did we learn?

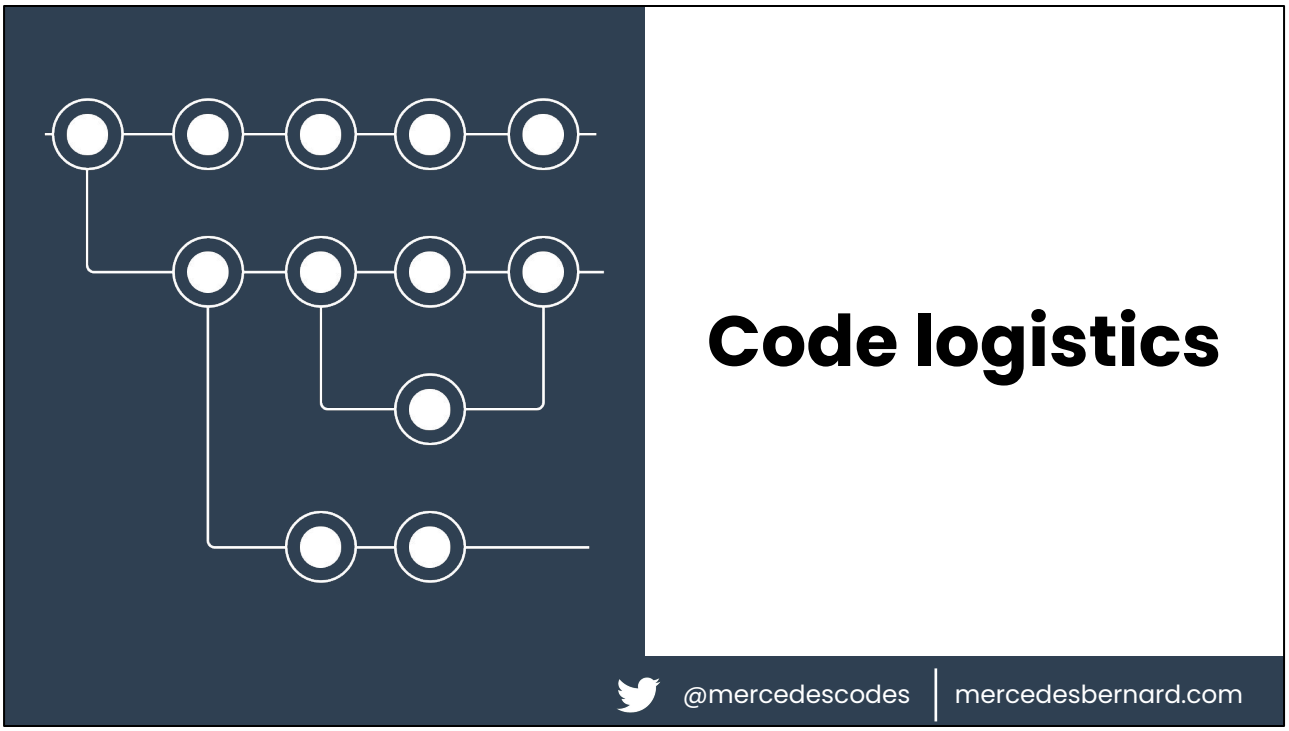
Delete unused code

@mercedescodes | mercedesbernard.com

I mentioned unused code while we were auditing but I learned about the maintenance cost of it the hard way.

If the code serves no purpose, delete it. You have source control if you ever want to revisit it. Thank it for its service and then let it go. Keeping a lot of unused code around can increase regressions, your need for test coverage, and bits of the code that would have to be touched if you refactor associations, method names, etc.

There is no good reason to keep this extra cognitive load in the code base. If no one has to know what it does because it’s never used, get rid of it.



As your team is working through all of this stabilization work, you want to make sure you're keeping track of your code logistics. Which branches are running on what version? Have upgrades been merged into feature branches? When was a database migration deployed? Was the migration rake task run? Having frequent, small deploys makes this easier. Long-running branches makes this harder. But we don't live in a perfect world so make sure you are paying attention to these things because a long-running branch is inevitable at some point.

The worst thing that could happen is that you don't realize data hasn't been migrated before you deploy the next migration that drops that table. Then you have to rollback and restore your last database backup. Not the end of the world, but definitely a little stressful and not super fun.

Final takeaways

@mercedescodes | mercedesbernard.com

We should try our hardest to prevent rescue missions. Technical debt is a reality of working with software and like any investment, sometimes it makes sense to incur some tech debt in order to achieve larger goals. As long as the investment is intentional and made with full awareness of the cost, then it can be a good decision.

Prevent future rescue missions

- Always leave the code better than when you found it
- Clearly name functions and variables
- Make use of service objects
- Add foreign keys and database constraints
- Use model validations
- Always add tests for new functionality
- Backfill tests for existing code that you're using in new features

However, we shouldn't be cutting corners and racking up tech debt willy-nilly. Advocating for code quality can be a challenge when there is pressure from business stakeholders to always be shipping but the alternative is worse. Advocate for quality software using shared language that aligns with stakeholders' values: time, budget, future extensibility for new features, customer experience, etc. And then in your work, continue to make small incremental investments in your code base, like clear naming, liberal use of service objects, backfilling tests, etc to prevent it from needing to be rescued later.

We can audit everything, make a plan, do the work, and sometimes the business will ultimately decide that it's no longer worth the investment. Or maybe the team just runs out of allotted time due to unforeseen challenges. It's ok. You did not fail. The application got into this place without you and you didn't cause it to fail.


Thank you!
@mercedescodes
mercedesbernard.com