3 Kinds of Tech Debt

Jul 10, 2021 14:54 · 1218 words · 6 minute read

In software engineering, we’re often faced with trade-offs that sacrifice what is best for the long term against something that increases velocity in the short term. A commonly used financial analogy is describing this as taking on **technical debt**.

I will not cover accidental and deliberate tech debt, neither will I give a decision making framework in this article. That is its own subject, and context specific.

However, there is a mental framework that has worked well for me across different teams, codebases and systems. I categorize technical debt in 3 buckets: **code, data, and architecture**. Let’s go over them now with some examples.

Code tech debt

This is probably the most common, and the first that pops to mind. This is debt that is taken in the form of suboptimal code that we commit. Some examples:

- copy paste some code for now and refactor it later
- write one big function that does all of the things
- import a huge library just for a tiny functionality requirement
- extend a function signature to manage an edge case

Everyone has likely seen a lot of this:

```
# This is a hack: @TODO FIXME
def barbaric_function(param1, param2, param3, param4, special_case=None):
    # This code is very brittle but we need to ship
    if special_case:
        import something.pretty.bad
        bad.hax0r(special_case)
    ...
```

Code tech debt tends to get called out the most, as it is more obvious when reviewing others' code, or simply reading codebases. It's the most "visible" of the 3 buckets. It is also typically easy enough to fix, since you can wrap your logic with tests and refactor, all in one code change, or a few surgical strikes (TDD has done wonders for codebases riddled with tech debt that need a makeover). Either way, it's just code, and that is a reasonable surface area to manage taking on and paying off debt.

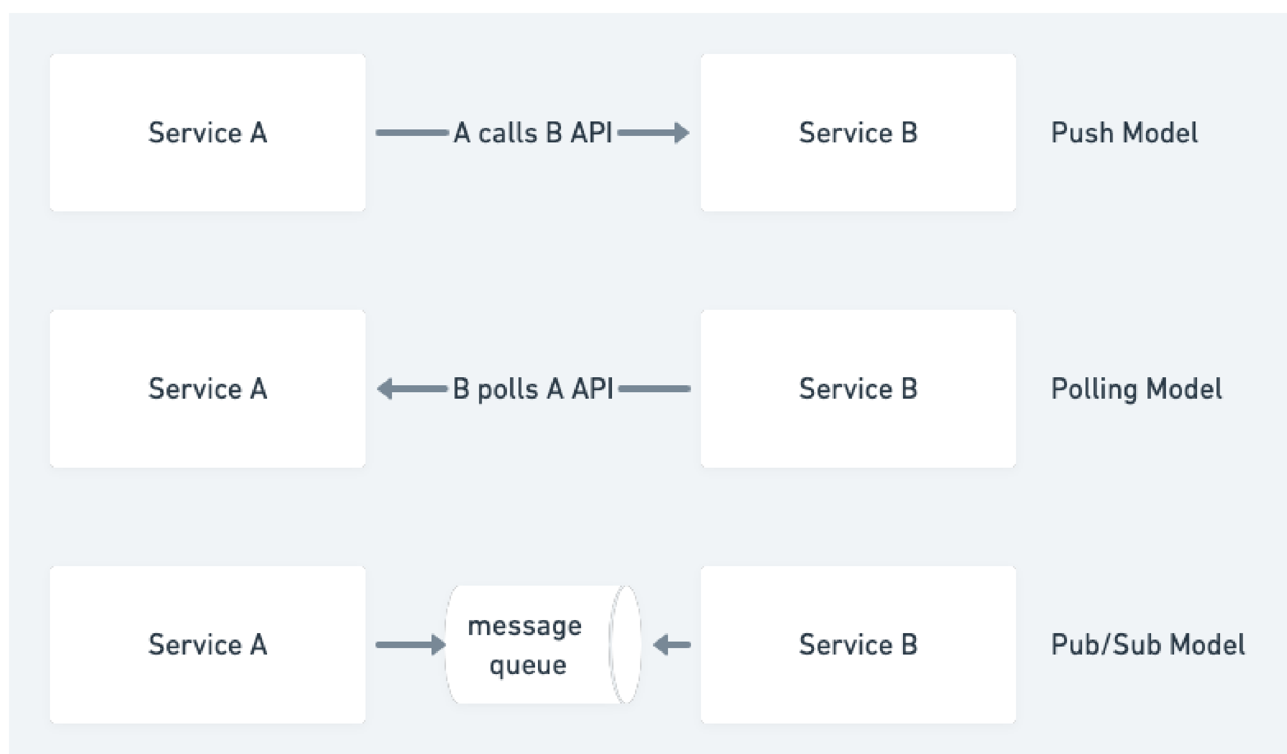
A lot of the time, when people say "tech debt" they are in fact thinking about this. I wish it wasn't so because this is the least harmful!

Architecture tech debt

This one is a much less common, but highly critical form of debt that projects take on. It has to do with design considerations around the system architecture, runtime choices, interfaces, service design, storage decisions, etc…

Some examples:

- What runtime the processes are running in?
- How are we message passing between processes / services?
- Are we adopting a monolithic strategy, sharing libraries, or a micro-services oriented architecture?
- Should this be an offline job or an online service?
- Will the workers be stateless or not?
- Should we adopt a polling model (pull), a trigger (push) or an event driven one (pub/sub) ?



These architecture decisions can drive projects into totally different directions. Taking the above example, using an intermediary pub/sub queue when traffic is smooth and QPS is low would likely add debt on the architecture in the form of maintenance and debugging. However if the traffic is very spiky and service B cannot handle peak load, then the push model would be the one that incurs the most debt in this area.

Paying back architecture debt happens typically on a much larger timeline. Long feedback loops are harder to learn from, and because of this architecture debt can go undected easily. The average tenure of a SWE in Silicon Valley is 18 months, but discovering if an early architecture decision was good or bad will likely take much longer! Uber discussing what happened when they hit 1k microservices and 2k engineers is the perfect example. Going the microservices route really improved their velocity in the short term, but incurred debt later for overall system debugability and latency down the line.

Architecture debt costs can be high (bad on-call schedules, rigid & hard to reason about systems, lack of debuggability), and are harder to pay back later compared to code debt. It is always worth analyzing deeply the systems you're working on, and understand what were the key decisions that were made – explicit or not – that impact the service characteristics (maintainability, flexibility, reliability, etc..). Some of these might be deliberate debt, and have had interesting consequences.

Finally, architecture tech debt can sneak up even when trying to avoid it. **Premature optimization** often leads to situations where we end up creating debt instead of the intent: avoiding it. An example would be a web service designed for 10k QPS, but ends up reaching 1/100th of that concurrency scale. Many architecture decisions could be different, and debt could easily be inserted as unecessary complexity.

Data & modeling tech debt

Data modeling seems less common than it used to be. There was a time where most engineers worked regularly on data modeling. Nowadays people usually delegate that to a special team ("the user service handles that"), or use tools that allow them to ignore the problem completely (graphQL, mongoDB come to mind in this area).

Data modeling can have impact both in the short term and the long term. In the short term, spending lots of time on models and types can feel very expensive, so it is tempting to pick something very flexible to optimize for early iteration and flexibility. However I've seen this go the wrong way (lets put everything in one JSON and do all of the filtering on the client, it's fine!), and ended up costing a lot in terms of backfills, data integrity fixes and remodeling efforts. Good data modeling has impact on both the code and the system architecture, so these 3 categories we are describing are not unrelated. However, data is one of the hardest things to get right, and also one of the hardest things

to change. This combo is the reason why data tech debt should be taken very seriously, and actively identified, called out, and justified. Down the line, when you want to change your data model, the dependency graph on such a change is often very unclear.It requires a combination of code changes, DB migrations, and backfills, all of which could have complex dependencies and intricacies, and could affect more than one system, team, or service.

I personally believe that this is the category of tech debt that happens accidentally the most. Understanding the read and write patterns, the relationships between your data models goes a very long way in preventing this kind of debt, or at least taking it on conciously. Anticipating where flexibility will be needed, and where it should not exist, are very subtle problems, that one gets better at with time, but has to do so conciously.

Conclusion

We've covered code, data, and architecture tech debt. Next time you're making design decisions or reviewing code, try and identify the debt you're taking on, and have an explicit conversation about it. What could be the consequences? Realistically, when if ever will it be paid off? Architecture and data tech debt will be the hardest to notice in the regular development day to day, but they have the toughest payback costs later, so they are worth your consideration.

Finally, I want to emphasize that **tech debt is not always bad**. Using MongoDB to prototype ideas or writing an ugly function that does abusive things to solve a critical bug can be justified, and the optimal way to do things. It's just a matter of doing so deliberately.

Discuss on hacker news

© Copyright 2022 nichochar

Powered by Hugo Theme By nodejh