Growing As a Mobile Engineer

ISBN: 978-1-63795-843-8 (ebook)

FIRST EDITION, v1.0

go.mobileatscale.com/growing

ISBN 978-163795843-8

9 781637 958438

Table of Contents

Introduction **5**

PART 1: Growing as a Mobile Engineer **6**

1. Map Out Opportunities to Grow 7
2. Typical Mobile Engineering Level Definitions 9
3. Professional Growth Versus Promotions 11
4. Mentoring 12
5. Changing Jobs 13
6. Down-Leveling When Changing Jobs 16

PART 2: Growing to Senior **19**

7. Master Your Main Stack 20
8. Get Familiar With the Other Stack(s) 21
9. Become More Product-Minded 24
10. Get More Feedback 25
11. Lead a Full-Stack Project 27
12. Ask For That Promotion 29

PART 3: Beyond Senior Levels **31**

13. The Challenge of Moving Beyond the Senior Level 32
14. The “Glass Ceiling” for Mobile Engineers 34
15. Go Broad or Go Deep 35
16. Understand the Business and Get Involved 36

17. Quantify Your Impact 38
18. Connect with Industry Peers 39
19. Public Writing and Speaking 40
20. Leaving Mobile Engineering to Further Your Career 42
21. It's Not Meant to Be Easy 45

PART 4: Mobile Engineering Management 47
22. Mobile Platform Teams 48
23. Mobile-Only Career Limitations for Managers 50
24. Advocating for Senior+ Mobile Engineering Roles 52

PART 5: Mobile Learnings From My Time at Uber 54
25. Platforms and Programs 55
26. What Good Mobile Architecture Looks Like 58
27. Hundreds of Mobile Engineers Working Together 60
28. I'm Actually Not an Android / iOS Engineer 64
29. Core and Optional Mobile Code & Modules 66
30. Mobile Oncall 69

It's All Software Engineering 71

Introduction

I have been building native apps for more than 10 years, six of these full-time. I was a principal iOS engineer at Skyscanner, then joined Uber as a senior Android engineer. I became an engineering manager, initially managing a team of five mobile engineers. Over the years, my team expanded from five engineers to close to 30, almost half of them mobile engineers.

This book is follow-up content Building Mobile Apps at Scale. While that book looks at engineering challenges of building large mobile apps, this book focuses on ways you can grow your career as a mobile engineer or engineering manager, while working on these kinds of apps and challenges.

Part 1, Part 2 and Part 3 share advice for iOS and Android engineers to grow to senior levels and above. These are the suggestions I gave to people on my team, and the approaches that helped them level up. This growth was both professional — people becoming more effective engineers — and measured in levels; people getting that promotion to the next level. At Uber, these levels meant promotions to the L5 (senior) or L5B (senior 2) levels, as well as working with a few people on crafting a path to the L6 (staff) level.

Part 4 is advice for mobile engineering managers. I have always found managing a group of mobile engineers to be seemingly more challenging, mostly as much of the business does not always understand the complexity of mobile engineering challenges at scale. For larger mobile organizations, you almost always realize the need for mobile platform teams, but how do you make this business case? And can staying in mobile slow your career growth in engineering management?

Part 5 collects interesting mobile learnings from my time at Uber. Uber is one of the largest mobile apps and mobile organizations in the world, with more than 300 native engineers, and apps that were the most downloaded application in the Travel and Food Delivery categories in 2019 (240 million downloads) and in 2020 (177 million). After working there for more than four years, I have insights to share that can be applicable at other high-growth companies.

Most advice in this book is more applicable for Silicon Valley-like companies. This means companies which work more like Silicon Valley-like companies and have a healthy engineering culture. It is companies that put parallel manager and engineer career paths in place, so that there are levels to grow beyond the “senior engineer”. The advice will apply far more to mobile-first organizations which invest heavily in mobile engineering, as well as to companies growing at a healthy pace.

PART 1: Growing as a Mobile Engineer

Regardless if you are just starting out in mobile engineering, or if you are years into this field, some advice is universal. Figuring out what growth at your company means, balancing focusing on titles versus gaining more experience, mentorship or changing jobs, are all this type of advice. In this part, we go through each of these topics.

1. Map Out Opportunities to Grow

What is the career progression at your company for engineers? How clear is this? At "good" tech companies, you find all of these elements:

- **A career ladder with levels and expectations** at each level. You can read several of these career ladders on Progression.fyi. Most large companies do not share these publicly, see the next section for an example on how such a ladder worked at a high-level at Uber.
- **A parallel individual contributor (IC) and manager career track**, where a senior engineer and a manager are typically of the same level and compensation. Engineers can get more pay and a higher level without moving into management, with tracks that lead to staff or principal engineering levels.
- **A clear promotion process** with clarity on how the promotion process works and how often this process happens. Most promotion processes include creating a promotion package reviewed by a group of managers, sometimes with senior engineers. The promotion package usually includes writing a self-review, nominating peers for promotion feedback, and the manager writing a review.
- **Getting regular proper professional feedback** on areas in which you can grow as an engineer. Better companies put in place feedback mechanisms where you not only get patted on the back, but you get constructive criticism about how you can grow better. What could you focus on more, to be a more efficient and better-rounded engineer?

If you have all of the above, figuring out how to get to the next level is a lot easier. Even so, as a mobile engineer, you can often find it hard to demonstrate the impact that the career ladder demands beyond the senior level. We cover how to spot and tackle this in the following sections.

If you work at a place that does not have this clarity on career, ladders, parallel tracks, or transparency in the promotion process, you will likely find both growing and getting promoted more challenging. A few things can still help with speedy growth:

- **Do you have a supportive and competent manager?** Someone who has been an engineer before and understands mobile development. Do they have a track record of helping people to grow and get promoted? Even if your company does not have all the clarity around levels or promotion processes, a good manager can help navigate this more ad hoc environment.
- **Do you get to work with one or more senior mobile engineers** who are both officially at a level above you, and you find that you can learn from? Are they approachable people who can mentor you? Having someone to learn from and a role model to work towards makes growing easier.

- **Do you work at a high-growth company, growing the mobile engineering team fast?** High-growth places can be disorganized early on, lacking much of the career structure that more mature organizations have in place. With growth comes opportunities to make a large impact, though. Many of these places tend to recognize and promote engineers driving this impact. The longer your tenure, the more you will be able to figure out how to make this kind of large impact. However, pay attention to whether you are growing professionally and learning from others, instead of just being the most impactful developer in a small bubble.

If you have none of the above, you will find it very difficult to grow or get promoted. In those environments, you will probably find much of the below advice of little use. You might also want to consider jumping ship to a company with a better professional environment.

2. Typical Mobile Engineering Level Definitions

At Uber, the engineering levels were defined as below. I am summarizing each level and in practice, each of the levels comprised three to four pages of definitions and examples, in order to bring the definitions to life.

- **Software Engineer (L3):** Accomplishes well-defined tasks, with guidance, where needed. Makes use of established best practices such as coding standards, tools, and processes the team uses. Eager to learn and try new things.
- **Software Engineer 2 (L4):** Independently completes projects within the team that could take up to a few months. Collaborates across teams and stakeholders in building solutions. Proactively addresses technical debt, issues, and inefficient practices. Helps more junior engineers.
- **Senior Engineer (L5):** Solves problems that can span multiple teams. Defines, plans, and executes impactful and challenging projects. Thinks beyond project boundaries. Identifies gaps in technology, tooling, or other areas and addresses them.
- **Senior Engineer 2 (L5B):** Owns long-term efforts, solving meaningful problems across several teams. Can go very deep or very broad in solving problems and creating solutions or platforms. Mentors and coaches engineers working on various projects.
- **Staff Engineer (L6):** Identifies, owns, and tackles strategically important problems at the company level. Delivers solutions that few others can, often orchestrating large efforts. Possibly someone with industry-wide influence and recognition.
- **Senior Staff Engineer (L7):** Forecasts and owns challenges the company will face in the coming years. Creates the vision of how to solve these, evangelizes, and executes on solving these. An industry-wide recognized expert in their area.

Levels at Uber did not have a mobile-specific definition, though it would have helped! Luckily, some other companies have put more detailed mobile pointers in place.

Mobile-specific leveling input is more rare in company career ladders. However, both Medium and Monzo have mobile-specific career tracks defined. Below are highlights from their levels:

Software Engineer (Entry-Level)

- Works efficiently with established iOS or Android architectures, following best practices
- Proactive in asking questions
- Builds simple screens or flows, fixes simple bugs or issues

Software Engineer (Intermediate)

- Makes minor improvements to the existing architecture
- Builds new functionality on top of the existing architecture
- Breaks down problems into steps
- Adds UI tests or snapshot tests

Senior Mobile Engineer

- Designs major new features
- Demonstrates a nuanced understanding of mobile platform constraints
- Builds complex and reusable architectures that enable best practices
- Owns and coordinates large scale architectural changes

Staff / Principal Mobile Engineer

- Coordinates group efforts
- Anticipates platform / project needs from other teams
- Industry-leading expert in mobile engineering, or sets the strategic direction for an engineering team
- Levels up engineers around them
- Attracts other very senior hires

Other engineering ladders that are worth studying are at GitLab, Meetup, Square, and Brandwatch engineering. Though all these ladders were written without a special focus for mobile, they should help you get a sense of what various levels mean at these companies and at similar companies.

3. Professional Growth Versus Promotions

It took me almost seven years to get that senior title after my first full-time software engineering job. I changed stacks multiple times, restarting my progress to learn something new. At the same time, I have known people who got promoted to senior two or three years after graduating who had far less exposure.

What matters more? The title or the experience?

By the time I was a senior mobile engineer, I had years of experience with multiple domains like web development, backend systems, native mobile apps, and thick clients. I have met seniors who lacked this kind of breadth but who had plenty of depth on iOS or Android. I have also met senior engineers who held a senior title, yet were barely familiar with engineering practices like testing or accessibility.

If you are playing the long-game in growing as an engineer, you are wise to focus on professional and not only titles. There are companies at which you can get a title without needing to showcase "baseline" skills for the same title, as at other places. However, if you keep growing your skills, you will often see a faster career progression in the long run, as you have a stronger foundation to build on. Growing your skills could mean going deep by becoming more proficient at good (mobile) engineering, as well as by going broad and developing your expertise outside mobile.

You have to decide how to balance going after the next level in the hierarchy, with your growth as a software engineer and problem solver. Focus too much on the title, and you could find yourself with a fancy one, yet unable to switch companies, and getting feedback that you lack the skills other companies expect from engineers at such levels. Completely ignore titles and levels, and you might observe peers who are less capable than you being promoted to more senior levels than you. They might ask for the promotion that you are not even mentioning, and your manager could assume you are happy with your current title.

Titles *do* matter at the workplace because it is these titles that help convey information about how capable and experienced an engineer should be. They also carry outsized importance for people who would have to otherwise spend lots of energy proving their competence. With a senior or principal title they can spend more time on the work and less time proving they are capable engineers.

Experience, deep understanding, and learning lessons first-hand matters just as much. Evaluate how important it is to focus upon titles or growth and decide where to focus. If in doubt whether to focus on titles or your growth, I suggest you focus on learning and trying something new. A title can be swapped when you switch companies, however it is a lot harder to upgrade your experience and skills.

4. Mentoring

How does mentoring work? I asked this question ten years into my software engineering career when I joined Uber. Until then, I had never been a mentor or a mentee, or at least never put this label on any of my activities.

Uber, however, had an official mentoring program. Almost every engineer I met had a mentor. Mentorship is an expectation for senior and above engineers and one that was listed in Uber's engineering competencies. Working at Uber, I found mentors, acted as a mentor, and observed engineers around me grow via mentorship.

Mentorship is a learning relationship between an experienced person and someone who wants to grow. With healthy mentoring relationships, both parties gain, not just the mentee.

Finding a mentor is a great way to grow faster. Most engineers assume you need mentors only when you are less experienced, but this is far from the case! At Uber, almost all mobile engineers had mentors and most of the new joiners were set up with mentors. Still, senior mobile engineers would often reach out to staff engineers they collaborated with loosely to ask for mentorship. The answer was almost always "yes"; both the mentees and mentors derived value out of this setup.

Some companies have formal mentorship programs and if you are lucky enough to have this, you can start there. Unfortunately, I have observed employers with this setup to be in the minority.

If there are no formal mentorship opportunities at your firm, you will have to reach out to an engineer you look up to and ask for mentorship. I suggest looking for someone at work and asking to set up a chat, and see if you can have regular catch-ups where you can talk about topics that can help you grow. Here is an introduction line you could use, and I have written about how you can think of setting up a kickoff meeting.

Mentoring someone less experienced is a fantastic way for you to learn. If you are already at a senior or above level, you could proactively look for engineers whom you might be able to help. Talk with these people and see if it is worth offering dedicated time or one-on-ones for them and if they want to take advantage of this. Mentoring someone helps you understand a different viewpoint. It will also push you outside of your comfort zone, as every mentoring relationship is different and you learn something new each time.

The most sought-after software engineers I know are all generous mentors. People not only look up to them for their coding, architecture, and execution skills. They also do so because they are approachable and continuously help others grow around them. This is because they are generous mentors, be this in informal or formal mentorship. And mentorship is how they keep growing their skills in areas like teaching, listening, leadership and growing a strong and supportive network around them.

5. Changing Jobs

Sometimes, the best way to grow is to start something completely new. This could be changing teams within a large organization, or it could be changing jobs altogether.

There can be numerous benefits to changing jobs. Pay is what most people think of, but that is only one metric. When you change jobs, you can gain a lot more:

- **Better compensation:** The obvious reason many people start job searching.
- **A more prestigious title**: Getting your first senior, lead, staff or principal title after a job change.
- **Expanding your personal network:** Working with a new group of people at the new company.
- **Forced to hit the ground running:** A new company is different from what you were used to.
- **Learn new technology:** The place you are joining will probably use different frameworks, architecture, and possibly even different languages.
- **Learn about a new industry:** Become familiar with how your new company makes money, who the customers and competitors are.
- **Relocate or travel:** Some companies offer or require that you move, which could be a great adventure. Other companies will give you opportunities to travel more, on the company's budget.

Having new opportunities find you passively is a great strategy, even if you are not looking. Knowing people in the industry, keeping your LinkedIn profile up-to-date, and writing in public are all good ways to increase your luck surface area of new, exciting, and rewarding opportunities to come knocking at your door.

Keeping an eye on the job market by browsing openings on sites such as StackOverlfow, LinkedIn, and other job sites, is also a good way to stay in the loop. If you see something that sounds interesting, consider reaching out and learning more.

When you are applying for positions, try to go through referrals, if you can, and update your resume. I wrote The Tech Resume Inside Out, a book with advice on how to write a good software engineering CV. The concepts are identical for mobile engineers, as they are for software engineers.

The typical mobile interview will vary from company to company. Parts that are fairly common to encounter are these:

1. Resume screen done by the recruiter or hiring manager.

2. Recruiter or hiring manager phone screen; a call that is typically not technical but often covers your experience.

3. Screening interview before the onsite one. Different companies choose different screening approaches.

3.1. Timeboxed coding challenge on the likes of Hackerrank or a similar service. You have a fixed time to solve coding challenges that are rarely mobile-specific. This challenge typically gets evaluated either by an engineer, or is done automatically. The least personal of screenings.

3.3. Technical screening call with a mobile engineer. You are typically asked about iOS or Android concepts and often asked to pair with the engineer to write code for a small problem.

3.3. Take Home exercise; an optional step many companies employ. You will most likely have to build a small app, such as in this exercise that JustEat asks candidates to complete. Areas you typically want to focus on are these:

- **Functionality:** Does your solution work as expected?
- **Edge cases:** Have you tested various text inputs, app lifecycle events, and other ways the app could behave in strange ways?
- **Documentation:** Did you document how to build and run the app, what it does, and what was out of scope?
- **Testing:** Have you written unit tests? Did you make the app testable?
- **Architecture and abstractions:** Did you choose a clean approach? Do components have single responsibilities?

4. Onsite interview: A series of interviews with engineers and the hiring manager. This is typically the last round standing between you and an offer. Each company will structure onsites differently, but there are main areas which most usually focus on:

- **Coding:** For companies that set a take home project, they typically ask you to extend this project on the spot, using your machine and setup. They confirm that you wrote the code, see how you respond to requirements changes, and how hands-on you are with mobile development. Other companies might ask generic data structures and algorithms challenges.
- **Mobile systems design:** You are typically asked to design a mobile app. While most of the discussion will be on the mobile architecture, you probably have to touch on what the API endpoints should look like and what backend assumptions you make.
- **Behavioral / hiring manager interview:** The least technical interview, where the hiring manager gathers information on a wide range of topics, such as what motivates you, how you work on teams, how you handle conflicts, and many others.
- **Bar raiser:** An interview specific to companies like Uber and Amazon. This interview is in-between a behavioral and a technical interview and typically involves a deep dive into one of your previous projects.

Preparing for the interview makes all the difference. Practice your coding, freshen up on designing mobile apps. Come to the interview with a mindset of focusing on learning something new, instead of stressing about the outcome. Here is a video with my advice on doing well in the interview from the perspective of a hiring manager. If you are planning to interview, give it a watch.

6. Down-Leveling When Changing Jobs

I was a Software Engineer 2 at Microsoft / Skype, then a Principal Engineer at Skyscanner, and then a Senior Engineer at Uber. All companies had the "levels hierarchy" as Software Engineer 2 → Senior → Principal. As you can tell, I was jumping between these titles.

I up-leveled at Skyscanner, skipping through the Senior level, straight to principal. Then I down-leveled at Uber, going back to a Senior level. What happened?

From my own perspective, I kept going up. At Skype / Microsoft, I had been at the same level of Software Engineer 2, for more than two years. I probably could have been promoted if I *really* wanted it. However, I never asked for a promotion nor even raised it with my managers. In hindsight, this was a mistake. The first thing you want to do to get a shot at promotions, is to ask for that promotion.

I also switched teams midway, from the Skype for Xbox One team to Skype for Web. Team changes often set back your path to promotions, as you need to build up trust with your manager and demonstrate to them that you are performing at the next level. I was the new person on the Web team and had to get up to speed with modern web development practices. By that time, I was an expert in C# and on Xbox apps. I also built highly-rated Windows Phone apps downloaded by millions of people, but I did so outside of work, meaning it did not count toward my work achievements or promotions.

I got the principal title at Skyscanner due to a combination of luck and relevant experience. It was luck that the recruiter at Skyscanner knew me from Skype. I had relevant experience of building successful mobile apps, while also having the experience of shipping large apps at a large company. Skyscanner wanted to hire an entrepreneurial person with mobile experience under their belt, who could join a new acquisition as the first mobile hire. My experience fitted the bill, even if my title at Microsoft might have not reflected this potential.

When I moved to Uber, getting a senior title did not feel like a down-level to me. Uber operated at a far larger scale than Skyscanner, and their compensation bands were higher as well. My senior offer at Uber was a significant bump from what I earned as principal at Skyscanner. Also, while I found it hard to find mentors to learn from at Skyscanner who were engineers more experienced than myself, I no longer had this problem at Uber. The engineers who had Senior 2 and Staff titles were people I looked up to and from whom I learned, from day one.

Many people find it hard to accept a down-level offer in title, even if it comes with an increase in salary. I have first-hand experience making several of these offers at Uber. For example, I extended one or two senior engineers at smaller companies Eng2 levels (L4), a level below the senior level (L5). The leveling decision always came down to what the company's definition was of a specific level, and whether we were able to determine that this person would *definitely* succeed at that level.

Hiring managers *want* people to succeed when they join a company. However, if at the interview, it is not clear that you will do well at a level such as the senior level, then it is common to give an offer at a level below, instead of just not making an offer. Especially at Big Tech, hiring and leveling are conservative. I recorded a video with more thoughts on why this is the case.

For mobile engineers, the biggest reasons for down-leveling — a company offering a title below your current one — is usually:

- **No real-world experience of building apps of similar complexity:** This applies especially to senior roles. At senior or above roles, you are expected to hit the ground running, taking ownership of parts of a complex app. If you have not done this before, you will likely need to learn the ropes. This is expected below senior levels, but at senior and above levels it is assumed you do not need much guidance. Much of this signal is gathered at the mobile systems design interview.
- **Not enough experience with engineering practices:** While it is understood that people learn quickly, if you are not hands-on enough with areas like testing, debugging, performance management, code reviews, designing clear abstractions, that other areas engineers at the company view as "everyday"; this will be seen as a flag. If there are only a few of these flags, you might get an offer at a lower level, with the expectation you will pick these practices up quickly in the new environment.
- **Not enough experience with activities expected within the company at the level:** For companies that have internal levels that are well-defined, new hires are often evaluated against this "ladder". If they are missing key skills or experience expected at that level, they might get offered a level lower. For example, at Uber, it was commonly expected that seniors would mentor less experienced engineers. If an interviewee with a senior title had no past experience mentoring — and could not describe good approaches on how to do this — they would be unlikely to get a senior offer. Many of these expectations will be internal to the company and, unfortunately, not part of the job ad.
- **A down-level would still fit your salary expectations:** If this is not the case, you would probably just not get an offer, in order to save time for both parties. This is another reason down-levels are more common for people coming from smaller companies, where the titled down-level can still mean a meaningful salary increase.

Either way, if you receive an offer, but the title is below what you expect, ask about the reason. I suggest talking directly with the hiring manager because they will have all the context, as the recruiter will most likely be trying to interpret the leveling decision.

When you have an offer extended, you *do* have the opportunity to ask to talk directly to your future manager before accepting. If the company setup is similar to that of Facebook, where you choose your own team after onboarding, you can still choose to talk to a hiring manager. If the recruiter rejects this, it is a major red flag and you should consider whether or not you are comfortable joining a company at which a hiring manager refuses to take time to talk. Getting on a call with the hiring manager is also a great opportunity to reverse-interview them before making an offer, a technique myself and others successfully used. Watch this video I made for ideas on what you could ask in a reverse interview.

My advice on accepting or rejecting such a down-leve" offer is this:

- **Do not expect to be promoted in a year:** No matter what recruiters tell you. The goal of recruiters is for candidates to accept offers. If you are only hung up on the title, they may often say things such as, *"based on your interview performance and experience, I am sure you'll be promoted within a year."* As an engineering manager, I am telling you the opposite. Even if you operate at the next level from day one, it can take a year or more to get a promotion, thanks to frequent tenure requirements.
- **Is it just the title that is stopping you from accepting an offer?** If you got the same offer with a different title, and you would accept it without hesitation, then perhaps you should consider accepting it? If the opportunity, the compensation, and the salary are all a step-up, do you want to reject it because of the title?
- **Would you be leveling up in terms of the engineering environment?** Is the app more complex than anything you have seen before? Is the mobile team larger than you have experienced? Is the company growing faster, with more opportunities opening up? If so, you might learn a lot more professionally, and this could be worth the switch.
- **Sleep on it and listen to what your gut tells you:** You will likely have far more information about the offer than you can put into words. These observations can manifest themselves in a "gut feel". Pay attention to this feeling. Are you more excited or more dissatisfied with the offer?

No one will be more invested in your career than *you*. Not the recruiter, not the hiring manager, not even your managers. Great managers will help on your career journey, but you cannot count on one always being there for you. Your career is far more than just a series of titles. Titles are like magazine covers; the cover of a magazine rarely does an accurate job representing what the contents will be, or if you will find the contents engaging and rewarding.

The longer you are in your career, the more you realize that the experiences, comradery, and challenges behind those titles are more important than the title itself. It will be up to you to decide how you choose your next adventures, shaping your career, one position at a time.

PART 2: Growing to Senior

After you spend a few years building iOS or Android apps and you feel you have become proficient at building apps, you may start to wonder how you get to that “senior” title and level.

The good news is this path is far more “beaten” than what comes after. Still, it is not always easy to get there. This part shares advice that can help you progress faster.

7. Master Your Main Stack

I have had the privilege to work with several excellent mobile engineers. They all shared two characteristics; they were experts in their own domain, but also "knew enough to be dangerous" in other domains such as backend, the "other" mobile platform, web, or data science.

I have seen several of these people grow from less experienced engineers, up to senior and above. Their journey was pretty similar within the company.

Early on, these engineers focused on going deep with iOS or Android. They were genuinely curious about what the platform can and cannot do, and were often enthusiastic users. They read the documentation, played with APIs, watched WWDC or Google I/O and pushed themselves to build the highest quality apps and components on mobile.

These people treated difficult bugs as a challenge and worked through them. iOS and Android engineers would track down if bugs were specific to frameworks shipped with iOS and Android and report these where they could. When using open source components, they would submit not just bug reports, but also file bug fixes. Much of this work was not expected and took additional time. At the same time, after finding the root cause of another tricky bug, these people grew more experienced.

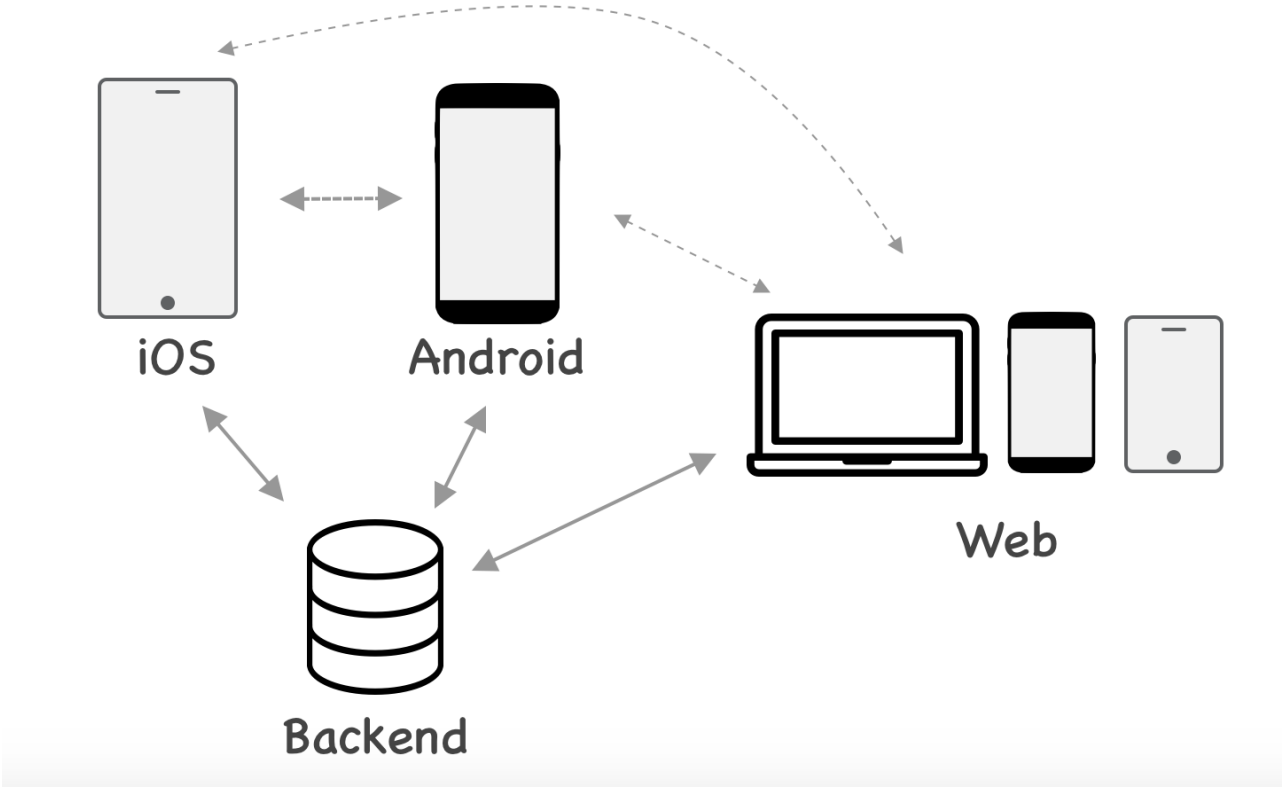
Covering all engineering aspects of the platform, not just the frameworks / APIs, was key to these engineers mastering the stack. They would get to a proficient level with testing, using debugging tools, performance monitoring, crash reporting, animations, analytics, and most areas covered in the book Building Mobile Apps at Scale. They would get to know a few iOS or Android frameworks really well, but only as well as they needed to.

These engineers would become experts and go-to people in a few of the areas. Areas I have observed included testing approaches, accessibility, UI animations, UI components, architecture approaches, or dependency frameworks. There can be much more. They would prototype and show to the team, propose work to be done and take the lead, and they would share their knowledge through internal or external knowledge sharing sessions.

Both iOS and Android change so fast that you can never know it all. But the good news is you do not need to be familiar with every last framework! After you are proficient with the stack and have covered most parts, you can decide which areas to deep dive into. More importantly, you can decide when to turn your attention to the other stacks.

8. Get Familiar With the Other Stack(s)

The most productive engineers were not content *just* knowing iOS or Android. Building mobile applications requires working with the backend team, and consuming and influencing the API endpoints. It is also helpful to know how the other platform builds their app for the same functionality. And in some cases, working with the (mobile) web team, who might build similar functionality, or could be further ahead, can also be beneficial.



Typical relationships between iOS, Android, web and backend teams. All teams depend on the backend, and teams often collaborate loosely with each other.

After mastering their main stack well enough, the standout mobile engineers started to spend more time with the backend, iOS / Android, or web teams. They would jump on opportunities to collaborate and learn. When they worked together on projects, they would pair, ask questions, and sometimes challenge themselves to make a small contribution on the backend, such as modifying an endpoint, or making a change in the web codebase. Even before contributing, they would read code reviews and join architecture discussions, even if only as listeners.

Productive mobile engineers heading towards senior levels or above, would almost always become familiar with how their other stacks worked. They got familiar enough to the point of collaborating meaningfully with the other teams.

Working with the "other" stack and coding *both* iOS and Android was a common sign that a mobile engineer was growing not just as an iOS / Android engineer, but as a mobile engineer. There were usually plenty of opportunities to do this, as most features we would build in parallel with each other.

These engineers would often take the initiative to facilitate iOS and Android planning in one planning meeting and one design document. They would then first start to passively review the other platform's code, then do this actively, and then ship a few small features on the other platform. In some cases, they asked explicit permission from their manager, but more often than not, they just did it. Their manager was always happy to see this. When I was that manager, I was ecstatic!

Influencing how backend APIs are built and implemented was another common way I observed these engineers bridge the mobile-backend gap. They would proactively start discussions about changing backend APIs, explain mobile challenges to backend engineers, and learn what was easy on the backend and why.

They picked up the backend "lingo" and became familiar with the main backend systems, and understood what happens behind the API layer. They would get why making a seemingly simple change such as introducing a new parameter to an API call, would be complex behind the scenes. They had a good mental map of the backend team's service dependencies.

There was only one secret to learning how the backend worked; they were curious, and they spent a lot of time talking with the backend engineers about how these systems worked. Backend code is also just code, so they often familiarize themselves with the codebase, architecture, and backend debug tooling.

Creating the opportunity to work with other teams, not just iOS, Android, and backend, but also with web, data science, product, and others, is something productive engineers proactively did. They would often spot an opportunity such as a new, upcoming project, and ask their manager to be involved. They did not just "wait for permission". They often started to collaborate on their own. You do not need permission to talk with data scientists, a web engineer, or any other person in a tech company. The best people I worked with rarely did and as their manager, this was something I not only appreciated, but encouraged.

An Android engineer educated themselves on machine learning, then paired with data scientists and proposed a product idea to automatically suggest default payment methods for users. They wrote a draft PRD (product requirements document) all on their own initiative, while keeping their manager — who happened to be me — up to date. In the end, we had to scrap the project for other reasons. However, they gained data science and product experience that made them a more valuable engineer.

An iOS engineer led a project with iOS, Android, and backend engineers to build a new payment method. The mobile part was pretty straightforward because iOS and Android just needed to integrate an SDK with a few tweaks. However, the backend was complex, as a migration was happening between two payment systems. The iOS engineer familiarized themselves with the backend stack. Also, they not only took part in backend discussions, but challenged the backend engineers and resolved deadlocks.

They approached the problem as a software engineer and the problems on the backend with a migration, were not that much different to those on the mobile would have been. They asked the right questions on options the team had, and the tradeoffs of each. When the team made a choice, they helped cut the scope to ship the first version. They helped put a testing plan together. And they paired with backend engineers, so when those engineers made an API change, they immediately tested it working on the mobile.

A great way to grow as a mobile engineer is to not pigeonhole yourself as "only mobile". You are a software engineer who can — and should! — solve problems where they arise. You will start on iOS or Android, but do not stop solving problems where these platforms end.

9. Become More Product-Minded

Product-minded engineers have lots of interest in the product itself. They want to understand why decisions are made, how people use the product, and love to be involved in making product decisions. They are people who would likely make a good product manager, if they ever decide to give up the joy of engineering. I have worked with many great product-minded engineers and consider myself to be this kind of developer. At companies building world-class products, product-minded engineers take teams to a new level of impact.

As a mobile engineer shipping a product that customers use, you are in a perfect spot to become a product-aware and product-minded engineer and deliver more impact. You already know the ins and outs of the user interface, and you probably work with designers and product managers. These are the very people you want to have access to, in order to become more product-aware.

Here are a few tips I have seen work well to grow your product-minded muscle:

- **Understand how and why your company is successful.** What is the business model? How is money made? What parts are most profitable, what parts of the company are expanding the most? Why? How does your team fit into all of this?
- **Build a strong relationship with your product manager.** Most product managers jump at the opportunity to mentor engineers. Having engineers be interested in product means they can scale themselves more. Before coming in and asking a lot of product questions, take time to build this relationship and make it clear to your product manager that you would like to get more involved in product topics.
- **Engage in user research, customer support, and other activities**, where you can learn more about how the product works. Pair with designers, UX people, data scientists, operations people, and others who frequently interact with users. If your company does user research sessions, ask to join!
- **Bring well-backed product suggestions to the table.** After you have a good understanding of the business, the product, and stakeholders, then take initiative. You could bring small suggestions to a project you are working on. Or you could suggest a larger effort, outlining the engineering effort and the product effort, making this easy to prioritize in the backlog.
- **Offer product / engineering tradeoffs** for the projects you work on. Think of not only making engineering tradeoffs for the product feature your team is building, but suggest product tradeoffs that result in less engineering effort. Be open to feedback from others about these.
- **Ask for frequent feedback from your product manager**. Being a great product-minded engineer means you have built up good product skills, on top of your existing engineering skillset. The best person to give you feedback on how you are doing at the product skillset is your product manager. Reach out for feedback on how valuable they see your product suggestions and ask for thoughts on areas for further growth.

10. Get More Feedback

What are your biggest strengths as an engineer? What are areas you should get better at? When was the last time you asked for feedback such as this, as opposed to getting comments at the not so frequent performance review? If you wait for feedback to be given by your manager and others, you will grow a lot slower, and you might not get the type of feedback you need in order to get better.

Proactively gathering feedback from your peers about how you can get better is a much more efficient way to grow. Here are various opportunities when you can ask for feedback and get more actionable responses, than if you simply ask your manager, *"Hey, can you give me some feedback?"*

- **Facilitate meetings:** After the meeting, ask someone else in the meeting to name one thing they liked about how you ran the meeting and one thing that they think you can do better.
- **Take part in meetings:** After a meeting that you felt you contributed well to, ask your manager or a peer how they saw your participation and value-add. What advice would they give you for next time?
- **Proposal feedback:** When you want to propose a new idea, for example an architecture change, new feature or something else, write up this proposal. Share it with a mentor, another engineer, or your manager and ask them what their take on it is, and where they would suggest changes or improvements.
- **Design document feedback:** When you author or co-author a mobile design document, ask for feedback from other engineers. Ask questions such as *"How well did you understand the idea? What areas do you think are less clear? What areas do you think I am missing?"*.
- **People reviewing your code regularly:** *"What is an area you think I could get better at with regards to my coding? Any recurring areas that you might have seen that you think I could focus more on?"*
- **Emails:** Emails are great ways to ask for feedback from your manager or trusted peers. This feedback can help improve how you write and communicate. Ask questions such as, *"Do you think the tone is right? Do you feel I get my point across well? What improvements would you suggest to make the email more crisp and professional? How would you have phrased the same message?"*
- **From engineers on your team:** "What is one thing that you think I am pretty good at, and one thing you suggest I try to get better at?"
- **From a product manager and other non-engineers you work with:** Ask them about how they view you and any advice they would give you to improve. You will be surprised by how good these observations can be.
- **After you complete a project:** Ask key engineers and stakeholders on the project, *"What is something you think I did really well? What is something you'd suggest I do differently on the next project?"*

- **Before a talk or presentation:** Ask one or two people to sit in on a rehearsal. Ask them to take notes and tell you where you can improve. I do this before important presentations and conference talks! For “smaller ones”, ask someone who will be in the audience. Ask them to be brutally honest, and make detailed notes, so you can improve further.
- **From your manager:** Ask them how they have observed you work on key projects, interact with team members, and other specific questions. Managers can forget to give feedback, even when they have observed something notable. Make it easier for them by being explicit in asking for this.

Ask specific questions to get specific feedback. People often struggle to give generic feedback when you ask questions such as *"can you give me feedback?"* Instead, ask questions that are more specific, such as:

- "What is the thing you think I'm the best at?"
- "What is one thing you would suggest I should start doing?"
- "What is something that if I stopped doing, I could perhaps be more efficient?"
- "On Project Zeno, what do you think I'm doing well, and what is something I could focus more on?"
- "I'd like to get better at {software engineering/working with other teams/proposing new approaches / other topic}. What is one thing you think I'm already doing well, and something you'd advise?"

Do not immediately push back on feedback that someone offers, even if you disagree with it. Doing so is the best way to stop that person offering any feedback. Is that what you really want? I suggest always thanking the person for the feedback as it is a lot easier not to give any feedback, especially to people who will not appreciate when others are blunt with them. This might also be why so few people offer feedback without you asking.

When you disagree with the feedback, do not leave it at that, though. It is okay to share that you do not fully understand the feedback and ask if they can offer more details or examples. Know that feedback is always subjective. Hearing feedback you disagree with is an opportunity to observe yourself from another person's viewpoint. Feedback is not something you always need to act upon, either; it will be up to you if you do anything with it after hearing it and mulling it over.

Good feedback is always based on trust and you have to build up this trust with others, in order to get the "unfiltered" feedback you need to hear. This happens one step at a time, both by you accepting feedback, but also you offering it and helping others.

I have noticed that most people do not get enough feedback and most company cultures do not encourage sharing of honest feedback. Early in your career, the lack of this feedback will not be visible, as you will probably keep growing just by learning new things. But later in your career, the lack of hearing what others would have to say, can hurt your prospects.

11. Lead a Full-Stack Project

I am a big believer in software engineers leading good-sized projects instead of project managers or engineering managers. By good-sized projects, I mean projects with up to around five engineers. There are so many benefits both to the lead and to the team. The lead is forced to understand all workstreams and get a handle upon the basics of project management. The team sees less project management red tape, and members get to help the lead, and learn on the way.

On my team, I rotated project leads on a per-project basis, so it would not be the same senior engineer leading every project. I consistently saw mobile engineers growing the most when they had to lead a full-stack project, with iOS, Android, backend and sometimes web, needing to work together.

Leading a project with several technologies forces you to understand how all parts work at a high level. Assuming you are an iOS engineer, you need to confirm that the Android plan makes sense and get a grip on what the backend team is doing. You could delegate all this work to the Android and backend engineers, however, that is sloppy leadership.

You want to understand what iOS, Android, and the backend needs to do. This way, you can look out for risks that can make the project slip, and help unblock each of the teams.

Mapping out and managing iOS, Android, and backend dependencies is the biggest learning that most mobile leads get when leading their first project. At larger tech companies, most project delays happen because of dependencies the team has. The backend team might depend on making a change to a service they do not own, or Android might be blocked on a bug in a new framework.

As an iOS engineer, you would normally blissfully ignore all the Android and backend dependencies and work heads-down on iOS only. However, as a project lead, you cannot avoid digging into what is happening, why it is happening, and how the team can resolve it promptly.

You should ask for the opportunity to lead a cross-discipline project once you are proficient with your main stack and have some familiarity with the other stacks. For better or worse, you will need your manager's support to get this opportunity and there is no guarantee that your manager is someone who believes in building a team where everyone is a leader. Still, if you do not ask, you probably won't get.

Instead of just asking to lead a project, try laying out the benefits this could bring for the team, yourself, and your manager. You can take an approach such as this:

"I'd love a stretch opportunity, and since I've already been working a lot with the Android and backend engineers, I was wondering if taking more of a lead role in an upcoming project would be an option? This could be a great opportunity for me to be more hands-on in helping the team ship and to take some project management load off the team. Just an idea, what are your thoughts?"

Feel out what your manager thinks about this approach. Trust is key in this case; as a manager, I would only entrust engineers to lead a project who I knew were reliable and would value the opportunity. Bring up your thinking in a conversation, follow up, and try to make a plan of how you can get involved with other stacks more officially as well.

12. Ask For That Promotion

I have written in-depth advice on how to get promoted as a software engineer and if you are close to a promotion, give it a read. However, the most important advice I have is this:

If you do not ask for or talk about that promotion with your manager, do not expect to get it.

So many engineers have been disappointed with not being up for promotion and yet most of them never broached the subject directly with their manager. Even if they brought it up, both parties did not end on the same page.

I once had an engineer transfer onto my team. I asked about how they think of promotions. They told me it was not their main priority, but they did want to get to the senior level when ready. I left it that way, making a mental note, "promotion is not a focus". Six months later, the same engineer was upset when they learned they were not up for promotion. In my mind, they were both not ready — and I had good reasons to think this — but I also assumed they were not interested.

Two people were at fault in this situation; me the manager, for not digging deeper and taking at face value what this engineer said. However, had the engineer brought up the topic of promotions earlier, I would have explained where I think they are growth-wise, and which areas they are missing to get to the next level. I have since learned my lesson, which is to bring up the topic of promotions early, to most people on my team. However, as an engineer, you do not only rely on your manager bringing this topic up.

Bring up the topic of promotions even before you want it. Gather information on how the process works, how your manager thinks about these, and what you can generally expect. A possible conversation starter could be this:

"Hey, I'd like to talk about promotions. Can you tell me how you understand promotions working at this company and how you approach deciding if someone is ready for a promotion? I know it could be early, but could we also talk about how close — or far — you see me from a promotion to the next level? I'd like to have realistic expectations and understand more on how this process works."

You need to have your manager on your side to be promoted. This is true no matter what the promotion process is like, whether it is ad hoc, a managers-only meeting, or a formalized promotion committee. If your manager is not fully supportive of your promotion, this will raise questions. Questions will lead to hesitation. Hesitation will lead to a conservative "let's revisit the next promotion period to confirm they are ready" decision.

Many people worry about this one: *"What if my manager does not like me?"* I hear you thinking. *"What if I have a bad manager?"* As a manager who has worked with dozens of other managers, some great, most good, some poor, you should not worry.

It is in *every* manager's interest to promote people who are ready. Promotions make managers look good as they show they have a team that is growing. In fact, the "poor managers" — managers whose teams do not trust them or have team problems — often fight even harder for people on their teams to be promoted, than the "great" managers who sometimes take a more conservative approach.

What "bad" managers may not do is set realistic expectations. Still, I have yet to see a manager who did not do what they could to promote people who were knocking it out of the ballpark.

If you have not had a direct conversation with your manager on promotions, the best time is now. Have that chat, even if it is just to explore what the process looks like, and signal that you *do* care about this topic.

PART 3: Beyond Senior Levels

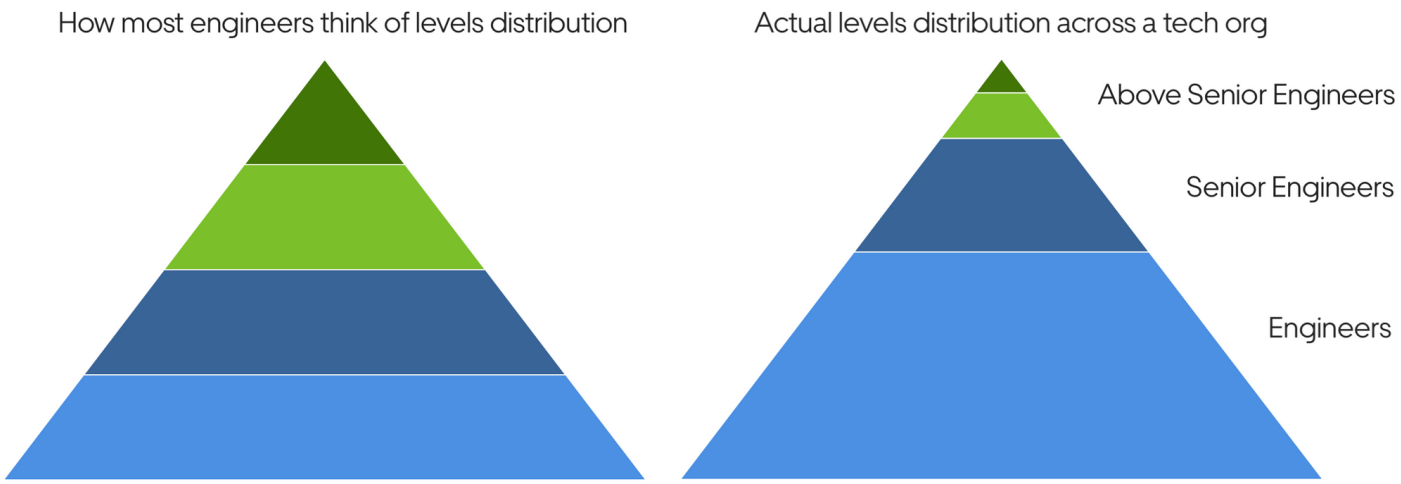
Once you have made it to the senior level, what is next? Assuming there are other engineering levels — such as staff or principal — how do you get there?

We'll cover why it can be difficult to progress beyond the senior title, especially as a mobile engineer. We'll also look at angles you can consider breaking what is often considered a glass ceiling in mobile engineering.

13. The Challenge of Moving Beyond the Senior Level

Growing to a senior mobile engineering level is usually pretty straightforward. By consistently doing good work and becoming an expert in your domain, be that iOS, Android, React Native, Flutter, or another mobile platform, you will typically get the senior title in a few years. It is not uncommon to see engineers with three to five years' experience and the senior title.

As you move up the "career ladder", it gets exponentially more difficult to get to the next level. This is because your ability to make an impact beyond senior level is something you have fewer opportunities to demonstrate. It is also because there are just fewer of these levels budgeted, or available, in any organization:



Actual distribution of engineering levels across an organization. There is usually a lower percentage of senior+ engineers than most people would assume.

When you have career ladders defined with Staff, Principal, or other Senior+ levels, figuring out how to get there is somewhat easier, as you have some reference on expectations. Career ladders can be a good starting point to discuss with your manager, evaluate yourself against, and to make a plan for how you can get there.

If there are no senior+ levels defined in your organization, this could make things more tricky. However, there are cases where the lack of levels might not be as much of a blocker of promotions:

- **If you are the most experienced mobile engineer with a senior title**, and there is no level above yours, this can be a good conversation starter with your manager. Is it time to define what the next level looks like and what it takes for you to get there? In the best scenario, you might help define the next level and could see yourself get promoted there later. In reality however, organizations that need to be reminded by their engineers about career progression, unfortunately, are rarely good at creating sensible career paths for engineers.
- **If there are other engineers with "higher" titles, but the definitions are missing** what those titles mean, you also want to discuss with your manager or skip-level manager about what that next level looks like, and how you can get there.

Budgets and headcount at the senior+ levels is something you have to become increasingly familiar with. At many companies, senior+ positions are treated similarly to manager positions, in that you only promote or hire into a new senior+ or manager position when there is a need — or a business case — for it.

A business case is usually an area with enough impact that it makes sense to invest more by hiring or promoting someone who is senior+, and therefore more expensive, but also brings skills that seniors do not have. For example, when a company decides to build a new payments system with an expected impact of $100 million in additional revenue, they might make a business plan to hire a team of 40 engineers in the next 12 months, 10 of them mobile engineers and one of them a senior+ mobile engineer.

14. The “Glass Ceiling” for Mobile Engineers

At most companies, mobile engineers often feel there is a "glass ceiling" limiting how high they can go up the ladder. I certainly felt like this at Uber, where despite building one of the largest mobile apps in the world, of the 10 to 15 engineers at the highest "level" (senior staff or principal), none were mobile. Of all staff engineers, only a fraction were mobile.

Most of the senior leadership, both managers and the most experienced engineers, had a backend or data science background. Despite the mobile app's complexity, I sometimes felt that there was more appreciation for various big data or distributed systems problems and the people leading them.

I do think that there is a "mobile-only" glass ceiling at many places. For any company, the iOS or Android app is one of the ways to reach business goals. These goals could be generating revenue, getting customers, and many other things. However, the mobile app is only part of the story. The other apps, the website, backend systems, data storage, security, customer support, fraud, and many other areas all play a role in reaching these goals.

Engineers who limit themselves to any single area will more likely feel they hit a "glass ceiling" impact-wise. Senior and above engineering positions are expected to deliver outsized impact, which is often only possible when multiple engineering disciplines work together.

The senior staff and principal engineers at Uber might have come from a predominantly backend background, but they had no problem working with mobile or web teams to orchestrate company-wide projects. As a mobile engineer, you need to do the same to grow beyond the "glass ceiling". You need to stop being "just" a mobile engineer.

You might find that in your organization, you can grow further only by being focused on mobile. However, if this is not the case, look for ways to either broaden your scope outside of just native mobile, or to deepen it. Broadening your scope is often an easier way to go. However, when deepening it, you could potentially get involved, and contribute to, things considered cutting-edge in the mobile industry.

15. Go Broad or Go Deep

For most senior+ engineer cases, I have observed these engineers go either very deep into the mobile domain or go broad. In either case, the impact of their work was visible to several teams, and often the whole company.

Going deep usually means doing impactful and foundational platform-like work. Deep work is typically possible at larger companies that solve mobile engineering problems that few other places can. For companies with mobile platform teams, much — if not all — of the deep work will happen in these teams.

Examples of deep work I observed at Uber included key work in creating the RIBs architecture, and rolling it out across several apps, and open sourcing it. Changing of linking of intermediate objects after the build to reduce the binary size during Uber's rewrite to Swift, creating a language-agonistic linting platform to be used across iOS and Android, or building a tool to understand Xcode build metrics, are all examples of work that I consider deep, rather than broad.

Mobile tooling or frameworks having gaps that make a large negative impact on the company, are usually good opportunities to step up to go deep. For better or worse, there is no general guidance on how you can demonstrate deep work.

Going broad is a more obvious strategy and one I often observe among senior+ mobile engineers on product teams. These engineers become familiar with several stacks, starting with iOS and Android, but they often expand beyond this stack. They then solve problems across multiple stacks.

Examples of going broad include architecting, coordinating, and shipping a major app feature, generating large revenue across iOS and Android. Another example is designing and shipping a mobile monitoring solution across iOS, Android, and the mobile web, coordinating the client-side implementation, the backend, and building the dashboards.

16. Understand the Business and Get Involved

Getting to a senior+ mobile engineering role is not all that different from doing the same on the backend, the web, or in another discipline. To get here, you need to impact the business in a way that goes beyond the senior engineer's scope.

Forecasting key business problems in a high-growth company and starting to solve them in a scalable way is one way of going about this. To do so, you need to understand what the business cares about, see what is getting in the way and take care of those blockers.

Experimentation is a good example. Fast-growing businesses typically care a lot about being able to move fast and try out new things. As the company grows, engineers typically start to add feature flags to roll out experiments. However, after a while, all these flags start to get messy. The mobile code gets littered with them and experiments conflict with each other.

An engineer who sees this problem ahead might start to solve it. It could be streamlining how flags are added. It could be creating a small framework for iOS and Android and only invoking flags through this, making it easier to track the flags later and to change the feature flag provider.

Shipping complex projects with massive business impact is another way to show results that are above the typical senior scope. These projects will be specific to the company's roadmap. At Uber, rewriting the Rider app or building tipping were both massive projects with business impact in the hundreds of millions of dollars per year. Dozens of teams and hundreds of engineers were involved. They were also great opportunities for mobile engineers to step up and lead respective parts of the project, enabling shipping of these giga-projects.

Getting to the senior+ level is rarely about technical skills; it is about understanding the business and driving to realize large business impact. People at the senior levels are already proficient with how to write good mobile apps. Engineers at the senior+ level, however, understand what the most pressing issues are for the business. They come up with workable and pragmatic solutions, then ship these solutions, often involving larger groups of engineers.

To get this business understanding, here are common activities I have observed efficient senior+ mobile engineers do:

- **Have one-on-one meetings with product managers,** understanding what is on the current and future product roadmap. When there are major projects coming up, get a seat at the table for the product planning.
- **Keep in touch with key engineers from a variety of teams.** Get to know what teams are working on and — more importantly — which areas they may struggle with. Do you see inefficiencies spanning multiple teams that you might be able to help solve?
- **Have regular one-on-ones with your skip level.** Senior+ engineers typically work across several teams. You want to get an understanding of what is happening not just in your team, but inside your organization. Skip-level one-on-ones will help with this.
- **Mentor less experienced engineers in the organization.** You will get exposure to what other engineers are working on and also what they might be struggling with, via mentoring. It is also a skill that will help you become a better teacher, and you get to meaningfully impact someone else's career.

17. Quantify Your Impact

As you understand which metrics the business deeply cares about, start to quantify the work you do and the results you achieve, using these metrics. You do not need to do this for every small piece of work you do. However, you should be spending at least half of your time on initiatives that move the needle for the business. You also want to quantify what this impact is.

It is a good idea to keep a work log with projects you are actively involved in and ones you are helping with, alongside their measured or estimated impact.

Aim to understand the impact of a project before you start work, or get involved in one. Operating at the senior+ level always requires being strategic with your time, as there are more things you could do, than there is time in which to do them. Having a rough idea of how a project will move the needle will help you choose to spend valuable time on more important things.

Learn to estimate not only the direct business impact of projects, but also the impact on engineering productivity. How would you decide if Refactor1 or Refactor2 is more impactful? You should ask what happens if you do not do Refactor1, and the same with Refactor2. What about upgrading to the new framework? Bringing a third-party vendor solution in-house? Using a vendor for a common task, instead of maintaining your hackly solution?

Get into the habit of knowing *why* any project is important, both for you but especially for the business. Numbers are hard to argue with, so figure out which numbers the impact of a project maps to.

Quantifying the impact on the business is key to having both deep or broad work recognized. This is usually easier to do with the broad work, as the business impact can likely be measured in revenue, number of users, or some other key metric. For deep work, much of the impact is down to developer productivity gain and sometimes industry exposure.

Examples of the impact of projects could be:

- $X generated in additional revenue.
- Y monthly users using the feature.
- Expecting to gain Z new customers in the first six months (and later update with actual data).
- X% speedup in build time (amounting to Y developer months / years across the company).
- Crash rate decreased by Z%.

18. Connect with Industry Peers

It is hard to grow if you feel like the smartest person in the room. If you already feel like one of the most experienced engineers within your domain at your company, consider meeting people from the outside from whom you can learn.

Networking at conferences and heading to local meetups is a possible way to get this exposure. Grabbing a coffee (or a video call) with an experienced engineer you do not know is also a good approach. Cold emailing experienced engineers and asking for a chat works more often than you might expect.

When connecting with someone else in the industry, I find it fascinating to learn about their environment, challenges, and learnings. Topics I tend to explore are these:

- What is your story in tech? How did you get where you are right now?
- What are you working on right now?
- Where do you see iOS / Android heading in the next few years?
- What is one of the biggest challenges at your work?
- How does your current workplace compare to previous ones? What is better and what is worse about it?
- What is something you are excited about?
- Can you recommend a good book you have read and learned from?

The more you put into conversations, the more you get out of them. If you share interesting things you see in your environment, you will probably hear the same from the other person. With some people, you might be on the same wavelength and catch up more regularly after a first conversation. With others, it might just be a one-off. Either way, you can only gain from meeting others in the industry, so do it!

19. Public Writing and Speaking

An even better way to connect with industry peers you can learn from, is for you to be that better-known person, thanks to your public speaking or public writing.

Few companies make public writing / speaking a requirement for senior+ engineers. Still, senior engineers who can represent the company well in public will see this as a supporting argument in getting to that next level. Experienced engineers who are prolific writers and speakers can also be exposed to more inbound opportunities for principal and staff engineering roles. When hiring externally for staff and principal mobile engineers, external proof that you are an industry-wide recognized expert can come from writing and speaking.

The ability for software engineers to write well is an underrated skill. For senior+ roles at larger companies, you are expected to be a good writer. This is because writing is a tool to reach, converse with, and influence engineers and teams outside of your immediate peer group. Writing becomes essential to make thoughts, tradeoffs, and decisions durable. Writing things down makes these thoughts available for a wider range of people to read. Things that should be made durable can include proposals and decisions, coding guidelines, best practices, learnings, runbooks, debugging guides, postmortems and even code reviews.

Public writing is a great way to achieve multiple goals at once:

- **Improve your writing** skills by writing up your thoughts.
- **Help others** by sharing your thoughts. People reading what you write will get new ideas, confirmation, or challenge their existing beliefs.
- **Get feedback on your ideas** from a wider group than if you did not publish. Well-written content receives well-thought-out responses you would otherwise not get.
- **Grow your network** and channel interesting opportunities your way passively by your writing, reaching people who would otherwise be unaware of you.

I am someone who has benefitted from all of the above by writing on my blog. I have been sent emails from people sharing how articles helped them try out new approaches and improved their craft. People have reached out to connect, inviting me to conferences, with job opportunities, consulting, and collaboration requests. I have also received follow-up thoughts from other experts on some of the engineering topics, and connected with several of them. This book would not have been born if I did not write publicly and share a draft for early feedback.

More inspiration on how to start and why writing can help you grow:

- Undervalued software engineering skills: writing well by me
- It's time to start writing by software engineer Alex Nixon
- Writing is thinking by Boz, VP of AR & VR at Facebook
- The online writing roadmap by David Perell
- Writing is thinking: learning to write with confidence by Steph Smith
- The writing well handbook by Julian Shapiro

Writing communities that can help you get started writing, give feedback and keep you motivated:

- Blogging for devs. An email course on advice on how to start blogging. If you need a trigger, consider joining this paid community that I am also a member of. For a reasonable monthly fee, this community can help you be accountable and inspired for publishing. This is the most active community I know of.
- TechWriters community, founded by Will Larson. A Discord group where people share drafts and give friendly feedback and encouragement to each other. The community is quieter than Blogging for Devs.
- Rands Leadership Slack group is the Staff-Principal-Engineering channel. Though not a writing group, this place can be a good spot to get inspiration from senior+ engineers, and read or share related resources.

Public speaking about mobile engineering problems and solutions at conferences or meetups, is a great way to teach. It is also a fantastic approach to get more exposure to like-minded engineers and leaders.

Internal company events and presentations are a great way to start building this skill, beginning with presenting in front of your team. Looking at senior and above levels, you will benefit from presenting at larger events such as all-hands and group events. Obvious topics include sharing how your team successfully shipped a project, or doing an overview of a new piece of technology your team or organization has adopted. However, there is no shortage of what you can present — and teach — about industry best practices, internal tools, your learnings, and many others.

Talking at external events is a great way to gain exposure outside of your company and to overcome your fear of talking in front of others. It is easier to get started with smaller meetups, and as you gain experience, move up to apply to speak at conferences. Shawn "swyx" Wang shares good advice on applying for speaking at conferences and how to submit Call For Papers (CFPs) that get accepted, and Ali Spittel also wrote with advice on public speaking as a developer.

20. Leaving Mobile Engineering to Further Your Career

How many *very* senior software engineers have you heard of who *only* do desktop development (Mac or Windows) and nothing else? There is probably a fair number. However, there are probably far more very senior engineers who might have done some desktop development, but who are also proficient with other stacks.

Mobile is still a growing area. However, it is just one of the growing areas within software engineering. There are so many more; data engineering (data science, ML, AI, and others), AR / VR, web, backend, distributed systems, developer tooling, and game development. I could go on.

No matter which company you work at, years into mobile development, you can easily feel your speed of growth reducing. The APIs are familiar, almost too much so. You can almost anticipate the changes coming with the next OS release, and the excitement of watching WWDC or Google I/O is fading.

When mobile development feels like solving the same problems repeatedly, such as building familiar features or apps, it probably means you have mastered the challenges your current environment has to offer. I have seen this happen both with people at smaller companies and at large places such as Uber.

Switching teams within the company, but on a stack different to mobile can be a good way to grow significantly as an engineer. I have seen — and encouraged! — mobile engineers on product teams to switch over and do backend or web development on our team, or a team doing similar work to ours.

Switching "stacks" while staying at the same company has a lot of advantages:

- **You already know the business domain.** If you join a new company, you will need to learn how the business works and figure out how you can make an impact. You already know all of this!
- **You already know the people** and those people will give you a lot more help. They also know that you are no expert in the new stack and will probably help more, appreciating how you are learning something new.
- **Your mobile expertise is an advantage.** When you build new backend endpoints, you are able to test them in the app. If you make web changes, you can look at the mobile codebase to see how it was implemented there. And you can — and should! — keep doing mobile code reviews.
- **You are far more likely to lead a cross-functional project.** Once you master your new stack, you are an expert on both this stack and on mobile. This expertise will open up opportunities for you to lead teams with a mix of engineers on your new stack and mobile engineers.
- **Switching back is almost always straightforward**, and you often come back re-energized. I have observed about half of mobile engineers moving over to another stack, who return to mobile with not only more experience, but more energy and a broader outlook.

Switching companies to do more complex mobile work can be a solution to keep growing in the mobile field. This is especially true if you are working at a smaller company, with smaller teams, solving less complex problems. Mobile engineers joining Uber from smaller startups, mobile agencies, or more "traditional" companies grew significantly, as they picked up the toolset to build large apps and to do this quickly and reliably.

However, getting an offer from a company where mobile work is more interesting, can be difficult. These companies often look for a track record of building and maintaining complex apps; something you might not have. You being motivated to "level up" by changing companies and putting in the time to prepare for interviews, will be key. If you are determined to switch, put in the work to make it happen.

Doing more mobile platform work is an area you will naturally gravitate toward if you stay in mobile engineering, after mastering how to build small and large apps. If your company has mobile platform teams, you will probably find yourself looking for ways to transfer onto these teams. If the company does not have this, you are still likely to be building reusable frameworks across apps or heavily contributing to these.

Deep platform work tends to move away from pure mobile engineering. To make good platform decisions, you need to build solutions that are as robust and reliable as the ones shipping with iOS and Android. To do so, you need to peek under the hood of the components Apple and Google ship and understand how they work. You decide how deep you go; the options are there, all the way to the implementation details of the Swift compiler.

To build well-designed platform components, it will help if you are familiar with iOS, Android, the web and backend, and the tradeoffs each platform comes with. You need to decide if a functionality should live on the client, or if the business would be better off if it was backend-driven. You have to explore tradeoffs between native code and solutions such as React Native or Flutter. You could find yourself building code generators, IDE extensions, or tweaking build settings.

I have observed many respected mobile software engineers venture outside mobile engineering. Some came back as better-rounded engineers, while some discovered an area they saw more potential in and never came back. A few examples of where mobile engineers whom I know ended up:

- **Infrastructure:** A staff mobile engineer who several, widely-used mobile frameworks moved to work on software infrastructure (observability, distributed systems, bespoke workflow orchestration etc).
- **Web:** Another staff mobile engineer moved to build a web application with the goal of this web app being as performant as a native app. They shared how the tooling and build performance in the web world was eye-opening and a joy to work with.
- **Backend:** A long-time iOS engineer decided they want to "break out" of their iOS bubble after seeing how people moved stacks more frequently at Uber. They taught themselves Java, applied for backend jobs, and moved to a new company to do backend engineering.
- **Back to Android:** A talented Android Google Developer Expert (GDE) felt there was not much new for them to learn with Android and worked on the backend for a year, building distributed systems with Java. They realized they liked mobile more and came back to be a tech lead on mobile-backend projects.
- **Back to iOS:** An iOS engineer stepped in to help out a larger project with some web and a bit of backend work. After completing a nine-month project, they decided it was time to go back to iOS, doing so with a broadened appreciation for these other two domains.
- **Founder:** A staff Android engineer left the company to found a startup in the mobile performance space.

21. It's Not Meant to Be Easy

If you are finding breaking above the senior level hard - it is! Although this level does not come with a managerial title, it is almost always a leadership one. Just as it is hard to get that first engineering manager opening, for those wanting to go in that direction the first senior+ role does not come easy.

Getting to a senior+ mobile engineering role means that you are a technical leader in your organization. You have to prove that you deserve to get here, and there is hardly a universally applicable method that works across all companies.

Keep in mind that senior+ roles are new at most companies. The chances are your manager will not know what to expect from you in this role, and this might be true for your leadership team. It can be up to you to shape the perception of the role and prove that you provide real value to the company — and to mobile engineers — as an individual contributor. Educate yourself on what this role is, for which I highly recommend the books Staff Engineer: Leadership beyond the management track and my book, The Software Engineer's Guidebook.

Once you make it to this level, pull up others as well. If you are above senior mobile engineer, the chances are that you are one of the most senior mobile engineers in the organization. This is a privilege that allows you to create meaningful change that impacts many other engineers.

- **Become a partner to management**. You should see yourself as both a partner and ensure that you get a seat at the table for higher-level engineering discussions. Being in the room for these discussions will also help you spot opportunities you can create for others.
- **Sponsor other engineers** in your organization. Sponsoring means actively helping these people get visibility, interesting work, and helping them succeed. As a senior+ engineer, people pay more attention to your voice in the room. When you say, *"I think Sarah should lead this project"* or *"I want to call out that Sam has gone over and beyond",* these might feel like small things for you, but can create large waves for others.
- **Educate managers on the challenges** mobile engineers face and verbalize the complexity of the work they do to overcome them. There is a fair chance your management chain will not understand the mobile work's complexity and frequently think "it's just an app". This thinking can be career-limiting not just for you, but for the whole mobile team. As a senior+ engineer, it is your role to keep on patiently educating management and other stakeholders on the complex and impactful work mobile engineers are doing.

Do not stop growing, no matter how "fancy" your title. It will be more important than ever to find mentors and a peer group with whom you can safely talk through your struggles and figure out how you — and your organizations — can keep improving. Where is the industry headed? What initiatives should you champion, and which ones should the company pass on? How can you tell if you are performing at this senior+ level well enough and how can you keep growing?

PART 4: Mobile Engineering Management

If you are a mobile engineer at a senior or above level, you could get opportunities to move into managing a mobile engineering team. This was my path into management at Uber and Skyscanner.

If you are interested in becoming an engineering manager, consider bringing this topic up with your manager. You will need both their support and a suitable opening to make a switch within the company. Also, consider doing activities that can prepare you to eventually move into engineering management, like giving credit to others, helping the team, networking or mentoring.

As an engineering manager with many mobile reports, I have had a few learnings that fellow mobile managers might find useful.

22. Mobile Platform Teams

Once you build and ship an app, and the app becomes successful, mobile engineering becomes more complicated than most people would expect. The book Building Mobile Apps at Scale lists 39 areas that can all cause headaches, many of which do not have a one-size-fits-all solution. Most of the problems come from smartphone development being a relatively young field of about 10 years, which is still changing rapidly.

Sooner or later, you are probably going to find yourself noticing engineers or teams "reinventing the wheel". As a manager, you might have more exposure to spot pain points that could be better solved with a uniform solution by building a shared framework, improving tooling, or unifying processes.

The idea of setting up a mobile platform team will probably come to you if your area has around 20 or more mobile engineers working on one or more apps. You can probably identify areas that make more sense to be owned by a single team. But how do you go about this? Should you champion this cause?

I set up a mobile platform team at Uber, creating the Payments organization's first mobile platform team. Here is the advice I have on how to go about an effort like this:

- **Verbalize the problems this team could solve.** Talk about issues the company is facing today and what the impact of a platform team could be. Talk with numbers. In my case, our pain point was how it took months to ship a new payment integration at Uber, and the cost was about an engineering year per integration. And we had a long backlog of these.
- **Empower engineers to think about problems, solutions, and data.** Mobile platform teams are run by engineers. Lean on them to learn about the problems they see. Empower them to come up with platform solution suggestions and try to estimate the impact of these. Consider telling them, *"Imagine I had a magic wand to create a platform team that can focus on solving the biggest pain points we have today. What are these pain points, and what would the impact of solving them be?"*
- **Get informal buy-in from your management chain and product management.** You will not get the support or funding to create a platform team if your manager(s) and product manager(s) do not agree. Once you have the data, socialize the idea. Aim to do this at the decision-maker level; the people who will decide to allocate funding. In my case, this meant my manager, my skip-level manager, and the head of product for Payments.
- **Write a draft narrative and share it with key people.** Once you have verbal buy-in, put together a narrative of the team. Summarize the context of why this team should be created, the problems it will solve, ownership, success metrics, staffing, roadmap, and other relevant topics. Make it as specific as possible, using numbers and dates.
- **Get the funding.** At this point, you should have support from your management chain and a strong business case. If you get the go-ahead, congratulations; you can formally start the team!

- **Don't sweat if you are not funded**. Even with support from your management chain, you might not get funding for a variety of reasons. This does not make the business case any less valid. See if you can do more with less, kicking off a few platform efforts with your current team. If you can show impact and further convince your management chain of the ROI, you will probably be able to revisit funding later. Everything you have put together should only help with this.

If you are planning to build a mobile platform team and are interested in the platform narrative I used when creating a platform team, please reach out to growing@pragmaticengineer.com, and I will be happy to share this document.

23. Mobile-Only Career Limitations for Managers

I have observed several mobile engineers being promoted to tech lead, who then move over to manage the mobile engineering team as an engineering manager. Some of these managers have spent most of their professional career building iOS or Android applications. After a while, many of them arrive at the question:

"Does being a mobile-only engineering manager limit my growth as a manager?"

My observation is that it generally *is* a limiting factor, though there are a few niche companies that can be exceptions.

Being perceived as someone who can *only* manage mobile teams can easily limit your managerial career growth. Many first-time mobile engineering managers amplify this perception by resisting the opportunity to manage non-mobile engineers. This can turn into a self-fulfilling prophecy because someone perceived to not be comfortable managing non-mobile engineers, will not be given additional non-mobile responsibilities, and there are only so many mobile-only teams that an organization has.

Changing jobs also becomes more challenging. I have interviewed and passed on many otherwise solid engineering managers who made it clear they would only want to manage mobile teams. Even at Uber, a mobile-first company, we had very few mobile-only teams.

I firmly believe as a manager, you do yourself a favor by exposing yourself to non-mobile domains, even if your main expertise always stays as mobile. This means saying "yes" to managing non-mobile engineers, in fact, looking for this opportunity. Once you have successfully managed non-mobile teams or engineers, you will be far less likely to be perceived as a mobile-only manager.

When I became a manager, I started to formally manage five mobile engineers. Informally, I also managed three other backend engineers. I spent a good chunk of my time learning about our backend systems and helping the backend team. A few months later, it was natural that I became a manager of both teams. I later set up a web team and then a mobile platform team, each being a new and fun challenge.

There are many reasons you can benefit from going broader than mobile after you become a mobile engineering manager:

- **You keep learning about engineering.** All the best engineering managers I know are technical and keep learning technical concepts. There are few better ways to learn than on the job. I learned more about distributed systems while managing a team and building them, than I would have otherwise.
- **You build trust easier with other teams.** Assuming you manage a mobile engineering team, many of the teams you collaborate with will be in other domains. The more you know about these domains, the easier you can have in-depth conversations and build trust.
- **You widen your career options**. An engineering manager who has managed mobile, web, and data science teams is someone who will be more promotable into a manager of managers, especially in the domains they know. They will also find it easier to win a new position; see the story above about the mobile-only engineering manager who had fewer career options.

Being mobile-only is not always limiting, though. I know of engineering managers and directors who are heavily specialized in mobile and keep growing in their career. They are typically people who lead mobile platform teams at large companies, doing cutting-edge work. They frequently end up at organizations of 50 or more native mobile engineers. Their skills stay valuable as very few managers are both hands-on and knowledgeable about large mobile apps and platforms.

There are many parallels between growing as an engineer to senior and above, and growing as a manager to beyond the line-manager positions. In both cases, you typically have more career opportunities when you are not perceived to be bound to a single domain, be it mobile, web, or something else. At the same time, this is not a blanket rule. People who are deeply specialized can become sought-after, precisely because of their unique specialization.

24. Advocating for Senior+ Mobile Engineering Roles

As a manager, it is your responsibility to help ensure there is a growth path for mobile engineers within your team. This typically means ensuring that most senior mobile engineers have a growth path. You will probably encounter these common scenarios that can limit growth.

1. There is no next level for engineers. You might work at a place where there is no level above where your most experienced mobile engineer is. This could be senior, or perhaps it could be principal. You should work with your leadership group to determine if this is what you want.

If the terminal engineering level is the senior software engineer level, I advise you to create a parallel IC ladder. The website Engineering Ladders or the chapter on dual ladders in the book Become an Effective Software Engineering Manager are both good resources to start.

2. There is no next level for mobile engineers in practice. There might be an "official" next level, but in reality, you cannot see any mobile engineer making it there, thanks to requirements which engineers cannot meet. You should take time to understand why this is, challenge it with your management chain and advocate for the impact and complexity of the mobile engineering work.

The only acceptable excuse I can see for mobile engineers not being able to get a promotion, is the lack of impact of the work. If mobile engineering is only shipping tiny, incremental features that make no business impact, well, this is a problem that you as a manager, should tackle. Is there really no larger business opportunities that mobile can enable? If this is the case, why are *you* still here? And if not, why are we not working on those?

3. Mobile engineers do not know how to get to that next level. The most common problem I have seen is this one; that there is a next level, and there might be one or two mobile engineers in the organization at that next level, but not in your team. However, mobile engineers are unable to figure out how to get to that next level.

Do *you* know what it takes to grow to that next level? If so, coach and mentor engineers, identifying opportunities. If you do not have impactful enough opportunities, can you create ones that also increase your team's impact and ensure you do not run into budget constraints?

If you are uncertain how to help mobile engineers grow further, fix this! Connect with managers who have done this or with more senior mobile engineers. Get more involved in the mobile roadmap and identify mentors for the mobile engineers and yourself.

A manager is only as good as their team. If you cannot help your most experienced engineers grow, you will probably also stop growing. Would you want to work for a manager who is not invested in helping you get better? I know of few, if any, great engineers who would.

4. Budget and headcount constraints. You might find yourself in a situation where mobile engineers are performing at the next level, but there is just not enough budget to promote them. Or perhaps you have a promotion budget, but you have a ratio of senior to non-senior engineers enforced from above.

This problem is entirely yours to solve. If the impact of the team warrants it, fight to get that budget. You should have the numbers to prove how last year you generated more incremental gains than the cost of those promotions, and that next year you are going to do even more.

However, if your team does not have the impact it would need to make space for more senior promotions; work on making this happen. Collaborate with your management chain and leadership. Understand where the business is investing, and see how you can help in that area. Explore if it is time to rethink the team ownership, or take on more impactful work, while spending less effort on work that has little result.

PART 5: Mobile Learnings From My Time at Uber

I spent four years at Uber. I joined as a senior Android engineer, then moved on to manage the Rider Payments Program team. My team transitioned to become the Payments Experience Platform team, and I helped create the first platform-only mobile team within the Payments organization.

Notable projects I was part of include Uber's Rider app to use RIBs in 2016, the Driver app rewrite, and countless other smaller and larger features shipped, such as tipping, Venmo and various payment methods.

Building payment methods might seem like simple projects on the surface, but they become complicated at Uber's scale of hundreds of internal teams, thousands of services, a presence in more than 60 countries and millions of customers. I talk about why Uber is so complex in more detail in this video, and gave a presentation on the complexity behind implementing Google Pay in the Rider app in this talk.

25. Platforms and Programs

Uber was the first company at which every team was either a *program* or a *platform* team, including some mobile platform teams.

Program teams were the majority of teams and people and I estimate between 65 - 70% of engineers worked on a program team. The definition of program teams was a team organized for focused innovation and rapid execution.

“Program team” is a term unique to Uber. Most other companies call these teams Product teams and this is the term Twitter uses. Program teams had the following characteristics:

- **Long-lived** as opposed to just assembled for a short project. The teams were formed around a mission such as Driving Experience, Marketplace Efficiency, or APAC Growth. The teams owned a well-defined mission. I started and worked on a Program team for a long time; Rider Payments. Our mission was to provide a magical payments experience for all riders.
- **Cross-functional**. Each team had the staffing to allow it to execute its mission. In practice, this meant a mix of non-engineers and engineers on a team. It was common to have a product manager, a data scientist, and even an operations person on the team. Engineering was also cross-functional, with backend and native mobile engineers on most teams. Some teams had web engineers as well. Teams usually owned features in the Rider, Driver, or Eats app. To build and operate features, you needed to build and own backend systems and the client code.
- **Customers are *external***. Program teams would work directly with customers, shipping features they used. Customers were not only Riders, Drivers, Eaters, or Couriers, although they were the majority. Internal customers included Operations, Customer Support or Accounting, and other smaller business area customers such as Freight customers, Jump bike riders, and others which could also be these customers.
- **Focused on a *business* mission**. All program teams were focused on moving the needle for the business. If the business changed drastically, for example because there was no more need to focus on growth, then the team would cease to exist. Over time, Program teams did change. They usually only grew, but some areas such as Uber Rush, did get shut down after they failed to grow with the expectations that the business had.

Platform teams created and owned the foundations and technology building blocks that Program teams could use to ship faster and focus on their business problem. Programs are built on top of Platforms.

- **Focused on a *technical* mission**. Platforms are typically focused on technical goals like scaling a key area, achieving performance or availability goals, or building an easy to extend and maintain architecture that serves multiple teams and areas.
- **Specialized and rarely cross-functional.** As Platforms solve technology problems, it rarely makes sense for cross-functional teams to do so. A Platform team typically worked on one domain or stack. Most teams would only have engineers of a certain stack. Later on, some platform teams started to work with TPMs or product managers, but this was rarely the case early on.
- **Customers are *internal***. Most customers for a platform team are engineers of other program teams or, in more rare cases, people from the business also using the platform. This was also why lots of platform teams could just consist of engineers.
- **Consumed by multiple "verticals" (Programs).** Each Platform would have multiple customers. This was also a requirement for a Platform team to own any functionality. If a service or functionality would only be used by one Program, that Program should own it, not a Platform.

Programs and Platforms shared several characteristics:

- **Ability to create services or new components**. You do not have to be a Platform to create new services, modules, or reusable components. Many platform pieces started with a Program team building it, and it later either becoming a Platform team — if more teams started to use it — or handing it over to a Platform team when more teams started to use it.
- **You build it, you own it.** Whatever a team built, they owned the quality, the oncall, and making sure things operated. This meant both Platforms and Programs typically had oncalls. There were very rare exceptions to this.
- **Quality of work**. Regardless of which team you were on, the quality bar was the same. Your code was expected to be tested and the functionality monitored.

Mobile platform teams were the ones who built and owned common mobile components. Examples of these teams included a team that built and owned the RIBs architecture and common UI components, a team or a subteam owning the networking layer, one owning mobile performance, and one owning the mobile build pipeline.

Mobile program teams were the "regular" teams of four to 10 engineers, where about half the team would be iOS and Android engineers. Smaller program teams would have an Android and an iOS engineer, while larger ones would have up to three of each. My team — Rider Payments — had five to seven native mobile engineers, alongside three to five backend engineers, making it a larger program team.

The existence of program and platform teams had career implications, especially beyond the senior level. To get promoted, you needed to perform at the next level and show the impact that is expected at that next level.

- **Programs** made showing impact easier. Getting promoted up to the senior level (L5) at Uber was typically a bit more straightforward on a Program team, as you had many opportunities to ship small and medium-sized but impactful projects.
- **Platforms** made doing work that impacted multiple teams or a whole organization, easier. For Senior 2 and above, the expectation was to have an impact across several teams. For Staff, the expectation was to impact a whole organization.

As a mobile engineer, it became difficult to get promoted to Senior 2 and above, without doing mobile platform work. A common way to get to this level was to either move to a Platform team after becoming senior, or to do a significant amount of platform-like work, even when on a Program team.

Platform teams also typically had more senior engineers — Senior 2 and Staff mobile engineers — who were more rare on Program teams. The people who worked alongside these folks benefited from learning — and being mentored by — these experienced engineers. Come promotion time, they were also more likely to receive strong peer feedback from these people.

I helped and observed multiple mobile engineers get promoted to above the senior level. The cornerstone of all these promotion cases was mobile platform work; creating components and solutions that were used by multiple teams to deliver business value that was much larger than any one project would have been.

26. What Good Mobile Architecture Looks Like

Since transitioning into mobile development, I have had many discussions on structuring the mobile applications we built. Use MVC, MVP, MVVM, or VIPER? Another pattern? At Uber, when I started, the old iOS Rider app roughly followed Apple's MVC pattern, while the Android one was built on top of MVP.

The Mobile Platform team then came up with the RIBs architecture, which bears the most similarity to (B)VIPER. The team decided to rewrite the Rider app using this architectural approach in 2016.

RIBs was the "lightbulb" moment for me on what good architecture looks like. And it was not because of the architecture approach itself. The approach was logical, if not something I would call groundbreaking. The groundbreaking part was how this architecture was not just a document and a single example detailed in one document or blog post; It was a toolset of various parts:

- **Thorough documentation.** The RFC (request for comment) document describing the architecture, was one of the first go-to documents for most engineers to understand the architecture's motivation and structure. Later, dedicated explanation documents were written.
- **Code generation** for both iOS and Android to generate pre-wired RIBs components. This code generation tooling was there from day one and tweaked later on. Now, almost all RIBs modules were generated identically, and adding a new RIBs component took a click of a button.
- **Onboarding documentation** to learn and master the architecture. Worldwide, Uber had around 15 to 20 mobile engineers starting per month at the time of creating the architecture. Even more engineers transitioned from the old architecture to the new one. All of these people needed to get up to speed with RIBs. The Mobile Platform team created an onboarding course that could be completed as self-study; an optional extra for existing engineers, mandatory for new joiners. Most new joiners took about a week to finish the course. We later open sourced these materials.
- **Example projects.** Together with the onboarding documentation came small, example projects showcasing the usage of the architecture with easy-to-understand examples.
- **Reliable support for the architecture** from the Mobile Platform team. If anyone had a question, they could turn to the Mobile Platform team. The team had multiple office hours per week during which you could talk to or Zoom with an engineer. They also maintained an oncall rotation to make it clear which engineer could be pinged for questions.
- **Proactive outreach for feedback on the architecture**. The Mobile Platform team sent out regular surveys to get feedback on the architecture and its pain points. This was a practice most Platform teams followed at Uber in gathering information and deciding what to prioritize. In the case of RIBs, I recall some tweaks being made after the first round of feedback. There were plans for a v.2 of the architecture, which never materialized while I was there.

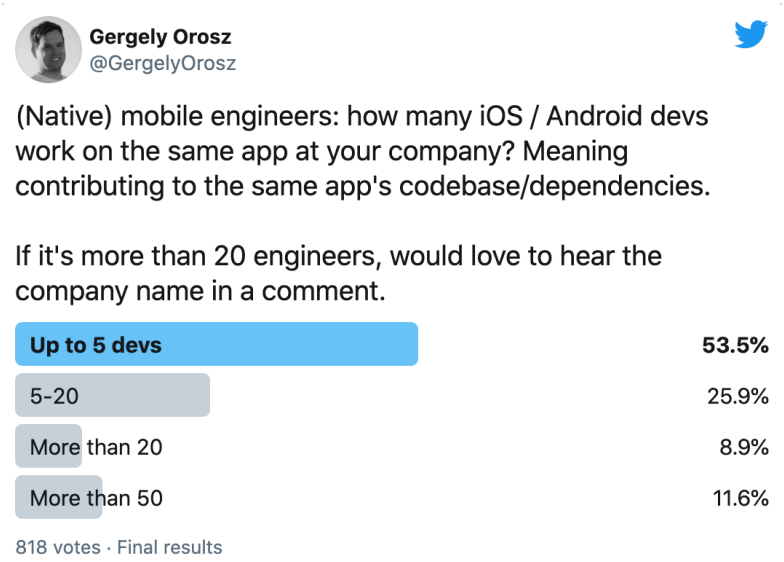
- **Tooling** to help to work with the architecture. The Mobile Platform team built an internal tool to visualize the structure of the application with nodes, making it easier to debug apps. You can see a demo of this tool in this video.

Good architecture is more than just architecture; it includes documentation, tooling, and support to adopt the architecture, work with it, and shape it to meet the needs of the teams. I am convinced that RIBs succeeded within Uber as the de-facto architecture because of this thoughtfulness.

RIBs is also one of the most starred — and probably one of the most used — architecture projects outside Uber. I am certain that open sourcing the documentation and code generation tools were what made this architecture so popular and easy to adopt.

27. Hundreds of Mobile Engineers Working Together

How many people work on most apps? According to a Twitter poll I ran, 20% of people work at places with more than 50 native engineers. At the same time, these companies tend to be some of the professionally most exciting ones to work, with compensation also often being higher than at smaller companies.



Results of a Twitter poll on the number of mobile engineers working on the same app

Uber had more than 100 iOS and over 100 Android engineers contribute every week to the Rider app, as we were rewriting it in 2016. To date, this has been the largest mobile engineering team working on an app I have seen or heard about. I would wager this puts Uber's app on a par with apps like Facebook and WeChat, complexity-wise.

So how do so many engineers work together, doing this efficiently? Uber did a number of things I had not experienced before:

- **Monorepo for iOS and Android.** Both iOS and Android moved to a monorepo in 2016 for performance reasons. Alan Zenino wrote up the story of the iOS monorepo, and Aimee Lucido shared the Android one.
- **Dedicated team for mobile CI / CD.** Uber did so many parallel builds on top of a monorepo that third-party build services did not make sense. Instead, a CI / CD team built custom pipelines and optimized builds using novel approaches such as speculation trees to create the SubmitQueue build system serving all of Uber's mobile engineering.
- **Dedicated mobile developer experience team**. A team of more iOS and Android engineers focused upon developer experience, ensuring that building, deploying, and testing the app is as easy as

possible for engineers. A dedicated team owned the in-house build infrastructure, and engineers worked on improving both the CI and local build performance.
- **A process for planning and discussing mobile changes**. Uber followed both an engineering RFC process for many years, sharing plans for mobile within the mobile engineering group, and also a PRD (Product Requirements Document) at the product level.
- **Changes that can be rolled back**. All changes going into the mobile codebase needed to have a mitigation plan in case an issue was found. We almost always did these with feature flags. For the few areas where feature flagging was not possible, such as library updates or major refactors, we followed a more strict planning and testing process.

Uber has been very open in sharing how these teams work together:

- Mobile architecture at scale; a talk I gave at Appdevcon
- iOS tooling at Uber by Alan Zenino, formerly an engineer on the mobile platform team
- RIBs: Uber's new mobile architecture from members of the Uber mobile platform team
- Failover handling in Uber's mobile networking infra: a deep dive into the mobile networking layer
- Mobile engineering articles on Uber's engineering blog

The custom mobile tooling was eye-opening to see at Uber. The company ran into the issue of most mobile tools on the market not working well, with hundreds of native engineers. So custom ones were built or used:

- **Phabricator** was used for code reviews and issue management for a long time. This was a tool built by Facebook, and Uber extended functionality in places. I understand the company is considering a replacement, after many years of usage.
- **Hotcakes** was a small tool for mobile hotfix workflows. Hotfixes; merging PRs to the release candidate, became very common at Uber's scale. I observed several of these for most releases. Each needed to be signed off by the manager of the engineer, and justification needed to be given to the release manager. Given the large number of hotfixes, we had levels within the hotfix process: Alphafixes for last-minute fixes and betafixes; hotfixes during the dogfooding period.
- **Morpheus** was the name of the experimentation platform, also used by mobile engineers. You could control your experiments and feature flags from its interface, rolling out to select app versions and segments, and monitoring results. See a write-up of under the hood of Morpheus / XP.
- **uMonitor** was our system used for alerting. Most of the uMonitor alerts were service-based and therefore backend driven. Mobile engineers had to be aware of which services impacted the mobile app and check if there was an outage ongoing, or if something was off with mobile. Read about details in uMonitor in this article.
- **Metro** was an internal tool for managing and visualizing the mobile release train. You could see which version of the app was at what status; was it in beta, or deployed to the app store? If deployed, what rollout status was it at?

- **Healthline** was a custom crash reporting tool. Think Crashlytics, but tailored to Uber, integrated with internal tools such as Morpheus, Phabricator, or Metro.
- **Canopy** was a manual sanity testing workflow system. While many of the release tests were automated unit, integration and UI tests, some tests still needed to be performed manually. Teams could record test instructions to Canopy, and a dedicated testing team would execute them after the build cut was made. In the case of failures, the owning team would be notified of this fact.
- **Rosetta** was our Uber-wide localization library, which iOS and Android also used. Localization was a beast at Uber's scale; hundreds of teams needed resources to be localized. Rosetta streamlined this process and integrated with Metro. After all, when a resource is not localized, the build should not be submitted to the App Store.
- **Commander** was a portal for reporting and tracking ongoing and resolved outages. When there was a problem with a mobile app your team owned, you would first check this portal to see if there was an ongoing outage.
- **Oncall dashboard** was a portal to track oncall pager alerts, annotate them and analyze shifts. Many mobile teams were oncall, and used this service extensively. See more details of Uber's oncall dashboards in this article.

Oncall dashboard used by all teams, including mobile engineering. See more details here.

Uber's engineering team grew to a large size at a time when there were still no good mobile tools to work with large codebases, or a large number of engineers working on the same codebase. While not all learning will be universally applicable to large companies in the future, one learning stands out:

When you are adopting a new technology or framework, the larger your team is using this technology, the more problems you will uncover and have to deal with. This is already true for new mobile languages (Kotlin, for example), frameworks (Swift UI, among others) and will continue to resonate in areas outside of mobile as well.

28. I'm Actually Not an Android / iOS Engineer

One of the Android GDE (Google Developer Expert) engineers on my team confessed, after a year of working at Uber:

"You know, Gergely, I thought I'd be an Android engineer when joining Uber. I'm actually not. I'm more like a RIBs engineer who sometimes gets to do a little Android work. I feel my Android skills are barely being used."

We both laughed, but there was a lot of truth to what this engineer said. Still, it can make little sense looking in from the outside; why would a large tech company hire expert Android / iOS engineers, then not utilize their knowledge? It is because of the existence of Platforms and Programs.

Companies like Uber try not to reinvent the wheel. They create Platform teams to own the common pieces of the infrastructure. Instead of each team deciding which HTTP library to use, and to upgrade this library on a regular basis, a Platform team provided the library and took care of upgrades.

At the scale of Uber, this decision made sense. The team owning networking measured performance, issues, and added support for use cases that a generic HTTP library would not have supported. It integrated with other in-house libraries such as Healthline or Morpheus. The library had built-in Thrift support, the protocol which Uber used on the backend. Uber got a better, more reliable, more performant, and more tailored HTTP layer than if it just used an off-the-shelf product.

The same happened with mobile architecture such as RIBs, Navigation, Experimentation frameworks, Data storage, Localization, and UI testing. All were owned by their respective Platform teams.

Any time you would reach for a solution that was generic enough that other teams would try to solve it, you would get a platform solution sooner or later. If you were a mobile engineer on a Program team, you had to become familiar with the Uber-specific frameworks and their limitations, more than the iOS or Android ones.

The heavy focus on the platform also meant that you would only be doing engineering "close to iOS or Android", if you were on a Platform team. Like many similar companies, Uber has an internal open source model. Even if you were not on a Platform team, you could still contribute to improvements of platform components, which all teams encouraged.

Still, I found many engineers were disappointed when they realized that they would not work directly with iOS or Android APIs, or that they would not use this knowledge day-to-day. And what does it make you, when you are no longer an Android or iOS engineer?

It makes you a software engineer.

A benefit that few engineers saw; at least at first, is how by not focusing on the iOS and Android internals, there was more space to focus on the problems themselves and other parts of the stack. Many mobile engineers ended up being a lot more involved in product decisions. Others started to learn the “other” stack. Some other people got hands-on with backend or web development, where, similar to mobile, you have to be familiar with many in-house frameworks.

What is clear, though, is that mobile engineering at a large company like Uber is very different from what most people expect, coming from smaller places. Like it or not, joiners have to adapt and find the upsides of a different setup.

29. Core and Optional Mobile Code & Modules

By 2016, the Android and iOS Uber apps generated around $1 billion in monthly revenue. This meant an average of $23,000/minute in revenue. To phrase this differently; for every minute one of the apps was down, for example, due to shipping a bug that crashed the iOS or Android app, the company could lose $12,000.

Backend systems at large tech companies often have a 99.99% reliability target. This means having an error budget of about 52 minutes of downtime for the whole year, aggregated. Using this approach for an app that generated so much business value seemed sensible.

We set the goal of the app working at least at 99.99% reliability. We defined working reliably as "the core functionality of the app working correctly throughout the user session." In practice, this meant that for every 10,000 sessions, no more than one could have issues on the "core" flows.

To achieve this goal, we first sat down to define what "core" meant. In the case of the Rider app, "core" covered the flow of signing up to Uber, logging in, requesting an Uber, taking and completing a trip. Over time, we added a few other flows, such as updating payments (as this could block taking a trip), verifying the profile, and other functionality related to the core business functionality.

What about functionality that was not related to this "core" part? For example, what if code related to updating your display name breaks? Well, as long as this is not "core", it is not part of this Service Level Agreement (SLA). Obviously, the team owning this functionality should make sure the feature works as well as possible, but you will not do additional monitoring or policing on these parts.

We divided modules of the app into Core and Optional categories. On top of the RIBs architecture, the Mobile Platform team built a Plugin system, where all "functionality" in the app would be packaged as a Plugin, which needed to be explicit about whether they were Core or Optional ones. The idea was that a Plugin could be remotely toggled on or off, so that if an Optional plugin started crashing, it could be turned off without impacting the app's Core functionality.

Taking the example of the app crashing when updating the profile information; the Profile Plugin would be an Optional one. If we detected a spike in crashes originating from this plugin, we could flip a feature flag on the backend and disable this plugin and flow. The team could then investigate and fix the root cause and ship the fix.

Core plugins were ones powering one of the Core flows. They needed to always work reliably. There was no option to turn these off, not even remotely. Turning off a core plugin would cripple the app. To ensure reliability, several quality gates were put in place:

- **Strict code reviews**. A member of the Mobile Platform team had to also sign off any changes to Core code, on top of the owning team.
- **Monitoring and alerting**. All Core flows needed to have first-class monitoring and alerting set up, with the owning team also being oncall for this functionality. As much of Payments was Core, we set up an oncall rotation to support this.
- **Thorough testing.** Any change to Core flows needed to go through additional manual testing.
- **Any change A/B tested** and gradually rolled out. Given that Core flows were directly connected to the business, any change needed to be A/B tested to ensure that trip-taking ratios did not change. Even a small bug fix in a Core flow could have unintended consequences.
- **Infrequent changes**. Core flows should change rarely. If a Core plugin changes frequently, find a way to separate the frequent changes in an Optional module that can be safely turned off.

Optional plugins were everything else, and the majority of the app. The above limitations did not apply to these plugins. Teams owning the plugin could decide how rapidly or cautiously they wanted to proceed with changes.

Still, even with Optional plugins, we followed the rule of all changes impacting production to be reversible, and did this by using feature flags. This went back to the scale of Uber, even for smaller features. Your feature might have only been used by one million riders a month, a very small fraction of the customer base. Still, if you shipped a bug without a way to revert the change, all those million users could have been impacted.

The core / optional setup worked well for a few reasons:

- **Forcing being explicit** about what were the essential parts of the business that needed to be 99.99% reliable.
- **Pragmatic balancing of quality gates overhead**. Making changes to Core code was slow and difficult, on purpose. Doing the same for Optional code was much easier. This setup forced teams to think carefully before changing the most critical parts of the application.
- **Mobile was proactive with reliability**. The mobile team set up, measured, and reported on these reliability goals earlier than many other parts of the company. During my four-year tenure at Uber, I saw high-profile mobile outages less frequently than on the backend or on the web. I do not have hard data to back it up, but I believe the explicit focus on Core flows being 99.99% available did pay off.

Downsides of the approach were fewer, but still existed:

- **Overhead and reluctance** in modifying Core flows. This was by design. Still, if your team owned a flow that was Core such as we did for Payments, the overhead became tedious. After several months of having to go through all the Core steps, we separated Payments flows to account for Core ones, leaving credit cards in this flow, and moving out regional or smaller payment methods into their Optional plugins.
- **Turning off Optional modules was less of a use case** than we expected. Although the plugin architecture was built to allow users to remotely turn off Optional plugins, we rarely did this. This was thanks to extensive feature flag coverage. When a plugin started crashing frequently, the owning team usually quickly found the recent rollout that caused this, and reverted via the feature flag.

Separating the core of the application and everything else is a worthwhile exercise and something I will do in future, following a similar approach to the one at Uber.

Defining the core flows makes it clear which are critical to the business. This clarity is both for engineers, but also for business stakeholders such as product managers and executives. Once these core flows are defined, it is easier to set the expectation that *any* modification to these flows will be *a lot more* time-consuming. The core flows should therefore stay small, but also be rigorously guarded and maintained.

30. Mobile Oncall

All of the 20 or so Payments iOS and Android engineers in Uber's office in Amsterdam, the Netherlands, were oncall. If you were the primary oncall, it would be your phone that buzzed when an alert went off.

It is not all that usual for native mobile engineers to be oncall. Before Uber, I did not experience native oncall rotations, and nor did many of the mobile engineers joining Uber. So who decided that mobile engineers should go oncall?

We decided, as a mobile team, to form a mobile-only oncall, from the early days of the Payments team and the Amsterdam office. Our team owned all mobile user experiences for paying in Uber's Rider app. In 2016, this meant that about $10 billion per year flowed through functionality we owned, and Payments was a Core mobile. By 2019, this number grew to $65 billion per year.

The backend team naturally had monitoring and alerting set up from the start. If a service was down, a backend engineer would be paged, and they mitigated the issue. But would backend alerts catch all possible issues?

We noticed outages that the backend team could not detect, or at least detected more slowly than if we had mobile monitoring and alerting in place. There were a few categories of these outages:

- **Crashes** happening on certain screens, for a subset of devices. The backend team had no way to detect this issue, and the drop in traffic to the backend was negligible. Yet this outage could still block a large number of users.
- **Networking issues** where the mobile client could not reach the backend, while retries could eventually go through.
- **Unresponsive pages** where users would tap, go back, and tap again. This could happen due to a screen "freezing", or a bug, such as a click handler not being wired up correctly.
- **Performance issues** where on certain devices, the app would be overly slow for a certain population.
- **Third-party checkout issues**. Several of our payments functionality was either powered by an SDK (such as the PayPal SDK) or used web views to direct customers to complete checkout flows. In some cases, we would see users getting errors from the third party and seeing error messages on their phones, but none of this information was forwarded to the backend.
- **Error messages** displayed on the phone that originated from the client, not the backend. We saw outages where a bug caused some of these errors to spike, yet our backend did not detect any of it.

We built a mobile monitoring and alerting system, similar to the ones we had for the backend. This system consisted of:

- **Mobile analytics mappings between iOS and Android**, so they mapped to the same business events.
- **A mobile analytics dashboard**, tracking both business events (user payment flow completed) and one-off events (entered credit card screen).
- **Alerting on critical flows** that raised PagerDuty alerts. Getting the sensitivity of these alerts right was tricky, but over months of iteration, we managed to get it at the right level.
- **Iterating** on monitoring and alerting at the mobile level, week after week. I found it challenging to get the granularity of alerts right, especially as many alerts would overlap with the backend team.

Mobile-only versus mobile and backend rotation for Program teams, was one of the biggest decisions each team had to take. Most of our Program teams were cross-functional, comprising 10 engineers, with three to five mobile engineers. In my experience, a rotation with less than five or six engineers is too heavy a load and can only burn engineers out.

A shared mobile and backend rotation was the choice that a few teams made. This came with the benefit that mobile engineers got more exposure to what was going on at the backend, and vice versa. However, most mobile engineers found it challenging to onboard to backend-heavy oncalls. The majority of alerts originated from the backend and mitigating them had to also be done there. In the longer term, these setups resulted in better knowledge sharing, better runbooks, and the teams bonding more.

A mobile-only oncall rotation was a given for mobile Platform teams, where the team was more than six people. For Program teams with fewer than six mobile engineers, this setup was only possible by merging similar areas into a larger mobile rotation, with eight to 10 engineers. My team followed this approach for quite some time. This setup was natural, as in 2016 we started with one mobile rotation across all Payments engineers in Amsterdam.

The benefit of this setup was the familiarity with alerts and mitigation. An added bonus was how mobile engineers had more opportunities to interact, even when on different teams.

The drawbacks started to show in the longer term. An engineer from TeamA would see an alert that made no sense. Investigating the root cause, they would discover that TeamB just released a new feature that TeamA had no context on. Initially, these incidents were one-offs. However, as the shared rotation became more "disjointed", they happened more frequently.

I strongly believe you should only have mobile oncall if you can mitigate potential outages on the fly, that is when you roll out features with feature flags, or have ways to roll back parts or all, of the application. When you have this capability, it is wise to invest in monitoring, alerting, and oncall. The setup of mobile-only versus shared oncall with backend, will depend on your environment, team, and other constraints. No two mobile teams had the same setup even within Uber and you should also aim to do what works for you.

It's All Software Engineering

I have been lucky enough to work across most native mobile platforms including Symbian, Windows Phone, iOS, Android, and plenty of others, such as web, backend, and thick clients. I have managed teams of mobile, backend and web engineers who all had the same question: What can I do to become even better?

In the end, mobile engineering is not all that different from software engineering, and the best mobile engineers have much in common with the best web, backend, data engineering and other types of software engineers.

Constraints, languages and frameworks are different from several other stacks. But the concepts, ideas, problems and solutions are more common than many engineers assume.

The best mobile engineers are usually great non-mobile engineers as well, at least, those who venture to other areas are. I owe much of my professional growth and outlook to not being stuck on just one platform, but exploring different areas, both as an engineer and as a manager.

Your journey will be your own; there are a lot of dividends to be gained by going deep in any area, and iOS and Android have plenty of depth. Keep in mind, though, that your career will be long; most of us will have 40 year-long careers. As you become more experienced, ask how are you thinking about growing your specialized versus your generic knowledge? What about sprinting quickly, versus pacing yourself? About choosing to go for that next title, versus taking on an interesting opportunity without that title?

Your career, your choices and your path will be unique. Good luck!

If you've enjoyed this book, please review it on Goodreads to help others discover it. Feel free to check out other books I've written, and to connect with me.

Thank you for reading this book, and if you have feedback or comments on the contents, you can reach me at growing@pragmaticengineer.com.

- Gergely Orosz