Short Essays on Engineering Culture

Luiz André Barroso

Over my years working as an engineer and executive at Google I have written a small number of short essays that focus on engineering culture. These articles invite the reader to think about how we innovate, what kinds of achievements deserve to be rewarded, and how to navigate technical decisions that affect large engineering teams.

Contents:

Roofshot Manifesto

We choose to go to the roof not because it is glamorous, but because it is right there!

I want flying drones that can bring me fresh produce. I am excited about contact lenses that measure blood sugar. I absolutely need human driving of cars to be outlawed.

Nevertheless I see us (Google) over-glamorizing the moonshot model of innovation, and I struggle to connect the 10X leaps approach to many of our successes. Rather, I contend that the bulk of our success is the result of the methodical, relentless, sustained pursuit of 1.3-2X opportunities -- what I have come to call "roofshots".

When I was a search software engineer I witnessed how our product continuously got faster and more amazing. It came from things like scoring changes, new click data, faster networking software, clever use of cheaper DRAM (later Flash), and so on. Each of them a roofshot. The combined result was awe-inspiring.

Our datacenter innovations followed a similar storyline. We started with a pile of machines in 3rd party facilities and decided we could do better. We kept improving electrical efficiencies, changing how airflow was provisioned, learned how to pre-fab pieces of the facility for faster construction and higher cost-efficiency, and eventually figured out how to nearly eliminate scheduled downtimes. Roofshot after roofshot and suddenly we have some of the most efficient and reliable datacenters in the world.

Seen from afar, these kinds of achievements could be mistaken as moonshots. They were, in fact, a sequence of roofshots. A sequence of roofshots is a compelling innovation model that can produce both quick returns and sustained transformative results(*).

Although moonshot leaps are rarer, particularly in systems areas, we must identify and seize them. So how do we find those rare opportunities? I subscribe to the artist Chuck Close's position that "inspiration is for amateurs; the rest of us just show up and get to work". Moonshots tend to reveal themselves to people that chase roofshots with passion.

So my summary advice in this manifesto is to go out there and have huge dreams, then show up to work the next morning and relentlessly incrementally achieve them.

--

*Math P.S.: A 1.3X roofshot per quarter is a clever way to achieve the equivalent of a moonshot in less than 3 years, with the added benefit of giving you a 30% improvement in the first 12 weeks! The area under the curve matters.

luiz@barroso.org
2014

The Eagle has Launched?

On the perils of mistaking a launch for an accomplishment

Why is it that so many more people remember the landing of the Apollo 11 mission on the moon than its take off? The Apollo launch burned 2 million liters of kerosene and liquid oxygen, generating 7.5 million pounds of thrust in a truly spectacular manner, but the landing of the Eagle module on the moon was the far bigger accomplishment.

Like the Apollo 11 launch, launching a product is a clear milestone that takes a huge amount of work and coordination by multiple teams. Think of some of our memorable launches -- launching Gmail on April Fools' Day, the Chrome comic book, and Psychic search -- these launches were natural moments of celebration.

But the catharsis of that moment can blur our perspective and lead us to miss a fundamental point: a launch is not itself a meaningful goal. It didn't matter that we launched these products successfully -- what actually mattered was that they were successful products that thrilled our users over time. Meaningful goals demonstrate material, positive impact on the utility of our products, the health of our business and happiness of our users. An overemphasis on launches incentivizes unripe products, fragmentation of efforts (too many arrows), and undermines the obsession to continuous improvement that is essential to truly excellent products.

We propose switching the language used for describing achievements from launches to *landings*. Landings occur once a measurable notion of success has been achieved -- happier users, delighted customers and partners, more efficient and robust systems. Landings become what we work for, what we celebrate, what we reward.

Landings, unlike launches, are not self-evident and require explicit definition and discussion. Do we measure success by growth in daily active users? Retention? Reduced tail latency? User satisfaction metrics? Profitability? How do these metrics differ across products (search has a different success metric than apps than platforms...)? Although it is not always obvious what the landing metric should be, forcing those decisions to be made early is essential to success. It is far more important to know "what does it mean to land" than "what does it take to launch".

Focusing on landings also makes it easier to have productive discussions about launch timing. Photos decided to postpone their launch by 6 months until they felt confident they had the right polish for a successful product. That was the right call: Delaying a launch can speed up landing.

Importantly, shifting our culture towards landings lets us highlight accomplishments that don't even involve introducing anything new. When the landing criterion is clear, it allows us to properly reward our individuals and teams who do work that improve existing products significantly, such as engineers that anticipate problems and actually simplify the solution during code reviews or teams that focus on critical systems, like Blaze, GWS or the Chrome updater, where landing means making important improvements (including reducing complexity) to existing systems. You can land without ever launching!

Let's ~~launch~~ land this alternative terminology for defining and rewarding accomplishments!

diane@google.com
luiz@barroso.org
2017

Innovate Within

The underappreciated art of non-disruptive innovation

In the early nineties computer architects believed the x86 instruction set architecture was on its last legs. As a then 20-year old technology, x86 (powering processors by Intel and others) appeared unsuitable to the needs of modern programs: its complex instruction set made it hard to run instructions in parallel and its 32-bit address space limited how much memory could be deployed. Intel had a solution: a from-scratch new line of microprocessors, named Itanium, with a new instruction set architecture focusing on instruction parallelism and extending the address space to 64-bits. Itanium was packed with bold innovative ideas, supported by massive investments, and it failed, while x86 products continued to lead the industry, and are successful to this day.

While this may sound like a story of innovation losing out to a legacy solution, what really happened is much more interesting than that. It is a story of innovation succeeding spectacularly, in large part because of the lack of disruption it caused to a huge and well established ecosystem. However daunting it seemed, teams at Intel and AMD continued to challenge the assumption that x86 was doomed and systematically addressed its technical limitations. They focused their efforts to *innovate within* their existing product line, and because of that, the vast deployed base of x86 software could immediately take advantage of those innovative ideas. Despite the buzz it enjoys in the tech world, disruptive innovation is rarely the best way to turn great ideas into successful products.

We work in a very successful, mature company. Unsurprisingly, many of our systems have an enormous user base, just like x86 had. Despite that, too often we choose to implement new ideas by designing a new thing from scratch, resulting in multiple systems solving similar problems. Besides this being inefficient, it also taxes our developers to navigate the resulting maze of options[1] when trying to build new products, leading to decreased product innovation velocity. On the other hand, implementing a new idea in an existing system will almost always yield greater return on investment – even if the constraints of the existing system might limit the headroom for the benefits of the new idea.

Here's an example: Imagine you have a new idea for a thing that could be twice as good[2] as the existing widely used solution if you implemented it from scratch, but only 20% better if you applied that idea to the existing system[3]. Assume your new thing will require as many people to maintain as the existing system, and that in the first 3 years you will capture at most 50% of the addressable market (optimistic parameters for an established technology area). The overall improvement will be doubling the goodness for only half of the use cases, so the net effect is only 25% better[4], while the total cost of the combined solutions doubles (two teams). Therefore, deploying the new idea from scratch has negative return-on-investment compared to not innovating at all, while deploying your idea in the existing system gives you a 20% positive return... To all your users... In a shorter time frame.

The assumptions in the math above are generous to the from-scratch approach. In practice, our enthusiasm to build our own new thing leads us to overlook how high the bar is for reliability, monitoring, security, privacy, data retention, accessibility, regionalization, user consent, efficiency, etc., in any new production-quality system. A reasonable rule-of-thumb is to only consider implementing a new idea in a from-scratch system if it is so great that well over 90% of existing users would be begging to migrate and take advantage of it right away. A from-scratch solution would also be the right choice for systems that have far outlived their ability to evolve but, as the x86 example indicates, we would be well served to focus our innovation on a new implementation that still supports the legacy interface, however imperfect that legacy interface might be.

Innovating from within, requires enlightened owners of existing systems to be open to trying new ideas, and benefits from a culture that invites non-team members to play in the existing system's code base. Good ideas should come from everywhere and not only from within. With the right engagement model, however, it can be the most effective way to maximize the impact of innovation in mature areas, allowing us to land achievements while skipping the launch altogether.

luiz@barroso.org
2022

[1] If a dev task needs to use 5 tools, each of them with 3 different versions, you have 243 distinct ways to complete that task.
[2] *Good* for the math in this example is a lower number. Think of lower latency.
[3] In many situations there is no penalty for improvements made in the existing system.
[4] Identical formulation as Amdahl's law

Three Lessons from Three Decades in Engineering

In 2020 I was invited to give an award lecture in which the awardee is asked to touch upon their career path. It was the first time I had to face myself as a writing subject and confess it was a bit uncomfortable. In the end I was able to summarize much of what I learned in conducting my career into the three lessons that make up the rest of this essay. I am sharing this in case they could serve as useful points of reflection for those earlier in their careers.

Consider the winding road

Our field, like many other technical disciplines, becomes richer and also more fragmented into sub-areas as our knowledge improves. There is always so much to learn about each of those branches of knowledge that many successful careers are built on continuing to go deeper and farther in a given area. An engineering or computer science education enables to follow that path but it also gives you a
foundation of knowledge that enables you to branch into many different kinds of work. Although there is always risk when you take on something new, the upside of being adventurous with your career can be amazingly rewarding.

I for one have never let my complete ignorance about a new field stop me from giving it a go. As a result I have worked in areas ranging from chip design to datacenter design; from writing software for web search to witnessing my team launch satellites into space; from working on Google Scholar to using ML to automatically update Google Maps; from research in compiler optimizations to helping deploy exposure notification technology to curb the spread of Covid-19[5].

It seems a bit crazy, but not going in a straight line has worked out really well for me and resulted in a rich set of professional experiences. If you choose that path, whatever the outcome, I guarantee you will be inoculated from boredom.

Develop respect for the obvious

The surest way to waste a career is to work on unimportant things. Having said that, it is far from trivial to know for certain when you begin working on a given problem whether it will turn out to have been relevant or not. I have found that many big, important problems have one feature in common: they tend to be straightforward to comprehend even if they are hard to solve. Those problems stare you right in the face. They are obvious and they deserve your attention.

Let me give you some examples by listing some of my more well cited papers next to the formulation of the problems address:

Publication	Problem addressed
ISCA'98: Memory System Characterization of Commercial Workloads With Kourosh Gharachorloo, and Edouard Bugnion	"High-end microprocessors are being sold to run commercial workloads, so why are we designing them for number crunching?"
ISCA'00: Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing With Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese	"Thread-level parallelism is easy. Instruction level parallelism is hard. Should we bet on thread-level parallelism then?"

[5] g.co/ENS

CACM '17: The Attack of the Killer Microsecond With Mike Marty, Dave Patterson and Partha Ranganathan	"If datacenter-wide events run at microsecond speeds, why do we only optimize for millisecond and nanosecond latencies?"
CACM '13: The Tail at Scale With Jeff Dean	"Large scale services should be resilient to performance hiccups in any of their subcomponents"
IEEE Computer '07: A Case for Energy-proportional Computing With Urs Hölzle	"Shouldn't servers use little energy when they are doing little work?"

If it takes you much more than a couple of sentences to describe the problem you are trying to solve, you should seriously consider the possibility of it not being that important to be solved.

Even successes have a "sell-by" date

Some of the most intellectually stimulating moments in my career have come about when I was forced to revisit my position on technical matters that I had invested significant time and effort on, especially when the original position had a track record of success. I'll present just one illustrative example.

I joined Google after a failed multi-year chip design project and as such I immediately embraced Google's design philosophy of staying away from silicon design ourselves. Later as the technical lead of Google's datacenter infrastructure, I consistently avoided using exotic or specialized silicon even when they could demonstrate performance of efficiency improvements for some workloads, since betting on the low cost base of general purpose components consistently proved to be the winning choice. Year after year, betting on general purpose solutions proved successful.

Then Deep Learning acceleration for large ML models arose as the first compelling case for building specialized components that would have both broad applicability and dramatic efficiency advantages when compared to general purpose designs. Our estimates indicated that large fractions of Google's emerging AI workloads could be executed in these specialized accelerators with as much as a 40x cost/efficiency advantage over general purpose computing. That success had expired.

With the help of the early Brain team we let go of our past successful CPU-only strategy and began building the hardware team that was to design Seastar, our first TPU. Almost ten years later TPUs remain a key differentiator for Google in ML-based products.

Conclusion

I should disclose that some of the lessons listed above only became clear to me when looking back so it would be problematic to claim that they really guided my journey but they are consistent with career choices I have made over the years. In the end, whenever I had the luxury of choices on what to work next I almost always picked the most fun, so it is possible that there is only one lesson to be learned here after all. I will close by saying that the effect of luck and the help of others on anything I've accomplished remains unwritten here, but they had as much of an impact as any efforts of my own.

luiz@barroso.org
2020

A Process for Company-wide Technical Decision-Making

We have a backlog of complex technical decisions that affect the whole company that we are slow to close or close them in a way that does not produce the intended outcomes. This happens because of the large blast radius of such decisions and the fact that many of our smartest people would disagree with any particular outcome. This document outlines a path for improving the process of making such high-level, sensitive technical decisions in a way that leads to effective execution on their outcomes.

When multiple intelligent people disagree on the course of action it is rare that you end up having to decide between a terrible option and a universally awesome one. How you make those decisions, on the other hand, really matters. First, notice that ***a decision in-itself accomplishes nothing***. The decision is a milestone that unblocks and lights up the path for execution. Excellent execution, in fact, can even turn imperfect technical decisions into successful products. If getting things done successfully in service of user needs (#landings) is what really matters, at the end of the decision making process you face the challenge that a substantial fraction of your team may disagree with the decision. The most powerful tool in your chest to engage them towards successful execution is the legitimacy of the decision making process. In short, we need to treat the process of arriving at those decisions as being about as important as the decisions themselves.

The rest of this document outlines a process for producing official Technical Roadmap Report (TRR) documents. These documents don't have to be produced for every decision but they will be used for the most critical or otherwise difficult technical decisions we face. We expect, in steady state, single-digit numbers TRRs every year, and for new TRRs to take between 1 month and 1 quarter to produce. This framework could, of course, be used for more localized decisions within teams under the discretion of their management and technical leads.

The goal is that such documents provide the official technical direction in critical areas that are rich in diversity of opinions, and they need to be disseminated in such a way that resourcing and prioritization for those areas across all of Google are based on them.

The process for producing TRRs will be as follows:

I. **Identify authors**[6] for each document. These should be technically senior and respected names in the areas. Ideally no more than five authors per document. Domain leads should be primarily involved in selecting the right authors for each topic and in many cases Domain leads will be co-authors but experts will be drawn from across the company.
II. **Identify key stakeholders/partners** that must be listened to before a first document draft is produced and have structured conversations so that they can contribute to the initial draft. Missing key stakeholders can be hard to recover from, especially if they will have a role in execution. Stakeholders are essential in producing findings that decisions are based on, for example.
III. **Use an appropriate mechanism** to keep track of all of the input from stakeholders such that a record of the rationale for the decision making process is kept for future reference[7]
IV. **Write a decision document** using the template outline below
V. **Review it** with the same group of stakeholders in II.
VI. **Publish it** to a broader audience still as drafts and provide a destination for a broader audience to and publicize them alongside a web form for feedback to be collected for suggestions for future revisions.
VII. **Schedule a date** to perform revisions. Yearly revisions are likely useful for most cases. Revisions could borrow from IETF RFC protocols, for example.
VIII. **Declare the decision as finalized** and work with the authors to help teams involved plan accordingly

[6] Product leadership should be considered for co-authorship
[7] A bug tracking system is one way of keeping a log of the feedback and evidence from various stakeholders

A common template for decision making documents is useful in building trust in the process itself. I propose we use the following structure for the documents:

1. **Goal** - describe what technical decisions are in scope and why they matter
2. Background - provide the minimum context so that people somewhat familiar with the area can follow the rest of the document
3. **Findings** - List the facts upon which the decision is anchored. Examples: what are the capabilities of existing systems, what are the gaps and overlaps among candidate solutions, what are the key requirements for successful solutions in the area today and over time, what objective metrics for landings in the area need to be observed, and any other factors that are pertinent to the decision. Findings ideally should be unambiguous and unanimously agreed upon by all the authors and preferably stakeholders.
4. **Recommendations** - The actual decision and its rationale is presented. Ideally this should provide both a northstar (what would a perfect world look like for the issues in scope) and a path for getting there. A time horizon for a "perfect world" should be 3-4 years, a timespan that is far enough from today to allow us to not be anchored by what is feasible this year but not too far into the future that our forecasting abilities begin to fade.
5. **Downsides** - Acknowledge the advantages of the options we didn't choose and the problems associated with the decision chosen, so that a reader can appreciate the diligence of the process.
6. **Follow-up Owners** - Identify the teams that are primarily responsible for executing on the decision, and work with the owners on planning and progress follow up. Since likely owners were involved from the beginning of the process (by design), this should not be a surprise to them. Execution plans should be filed no more than one quarter after the TRR is finalized, and they will be tracked by a central technical office (xGE in Google's case)

luiz@barroso.org
2021

Advice for Engineering Review Presenters

Engineering review presentations are a special breed of technical talks. Here is some advice on making those reviews productive.

Tell the audience why you are there

Eng. reviews can be a good forum for reporting progress, gathering design feedback, asking for management decisions, and communicating plans[8]. Make the purpose of your presentation clear before you start. Put a name / contact email on the title slide.

Be a spoiler

An eng. review is not a mystery novel. Do not hold back the goods until your 30th slide; present them on the 2nd instead[9]. Take a moment to describe the problem before the solution, and to explain why any existing solutions fall short[10]. Spend the rest of the review providing necessary detail, justification or background. Remember, you may never get to your 30th slide.

Think about to whom you are presenting

It takes a bit of effort to take a step back and look at your material from the perspective of someone that is not immersed in the subject at hand. That effort always pays off. If you are not sure, test with someone outside your area.

Internals are less important than interfaces

Engineers tend to focus on the implementations and internal details. Eng. reviews are more useful for communicating interfaces, assumptions, external behavior, or how your project otherwise affects other teams' plans.

Slides are bad documents; documents are bad slides

Verbose slides distract the audience, but telegraphic bullet points are of poor documentation value for those who miss the review. A useful technique is to make a first draft of your slides without worrying about verbosity, then move all the text to the notes (or a separate document) and re-write the minimum amount of bullet points necessary to script your talk. If you're using fonts smaller than 16pt you have too much text on a slide.

Time matters

Many unsuccessful eng. review presentations fail because of poor time management. Plan for questions and designate a team member to be your time keeper. Actively manage the discussions, flagging issues that are taking too long by writing them down for post-review follow-up actions. Practice the talk at least once, to make sure it fits into ⅔ of the allotted time (budgeting the other ⅓ for questions).

luiz@barroso.org
2013

[8] Try to anticipate the outcome of the eng. review. For example, a review communicating plans should not come as a complete surprise to the relevant technical leads and managers in your area.
[9] People sometimes avoid this tactic since it may cause audience members to jump ahead and ask questions that will be better dealt with by the later slides. Those questions are a positive consequence of this technique, so don't avoid it. You have grabbed their attention. Instead manage your audience and let them wait for the details.
[10] Ideally a new solution should allow existing ones to be deprecated.