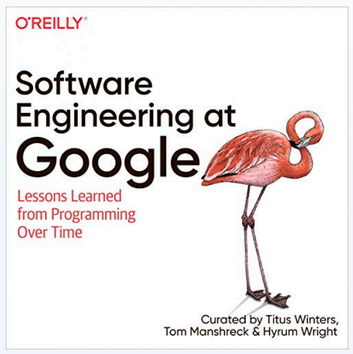
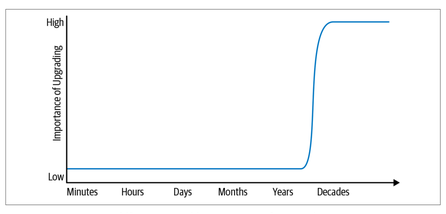
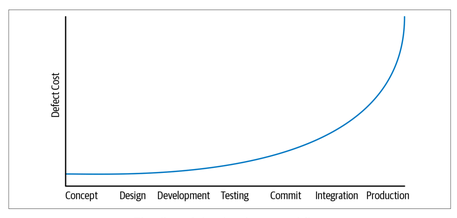
- Software Engineering at Google
  - 

  - 1. What is Software Engineering?
    - Preface
      - 3 fundamental principles
        - Time and Change
          - How code will need to adapt over the length of its life
        - Scale and Growth
          - How an organization will need to adapt as it evolves
        - Trade-offs and Costs
          - How an organization makes decisions, based on the lessons of Time and Change and Scale and Growth
      - 3 main aspects
        - Culture
        - Processes
        - Tools
      - 3 differences between programming and software engineering
        - Software engineering is program-ming integrated over time
          - Programming
            - just development
          - Software engineering
            - development
            - modification
            - maintenance
        - Time
          - What is the expected life span of your code?
        - Scale
          - programming task is often an act of individual creation
          - a software engineering task is a team effort
        - Complexity
      - Time and Change
        - Short life span of the code (no maintenance needed)
          - In education environment
          - In some industries - mobile or early-stage startups
        - Unbounded life span
          - Google Search
          - Linux Kernel
          - Apache HTTP Server
        - 

        - Life span and importance of upgrades
        - Hyrum's Law
          - ⚠️ With a sufficient number of users of an API, it does not matter what you promise in the contract: ALL OBSERVABLE BEHAVIORS of your system will be depended on by somebody
        - Example: Hash Ordering
          - 👎 Hash flooding attacks provide an increased incentive for nondetermi-nistic hash iteration.
            - ❓ Q: how such DoS attack is possible if hash order is undefined?
          - Potential efficiency gains from research into improved hash algorithms or hash containers require changes to hash iteration order
          - 👎 BUT Per Hyrum's Law, programmers will write programs that depend on the order in which a hash table is traversed, if they have the ability to do so.
            - ⚠️ If you don't know how long your code will live, or you cannot promise that nothing you depend upon will ever change, such an assumption is incorrect
            - There is a difference between "it works" and "it is correct"
              - 👎 "Clever" code can be a compliment in programming but it is an accusation is software engineering
                - 🙂 That's why I don't like lambdas, functional programming, higher order functions, ternary operators, and other language/framwork-specific shits... ;-)
        - Why Not Just Aim for "Nothing Changes"?
          - Because bugs (and necessity for changes) can be anywhere!
          - Backward compatibility ensures that older systems still function
            - but that is no guarantee that OLD OPTIMIZATIONS are still helpful
      - Scale and Efficiency
        - What means scalable?
          - 👍 Google's production system as a whole is among the most complex machines created by humankind
          - Your organization's codebase is SUSTAINABLE
            - when you are able to change all of the things that you ought to change, safely, and can do so for the life of your codebase
          - 👎 If costs grow superlinearly over time, the operation clearly is not scalable
            - Need to think about scaling
              - Human resources
              - Compute resources
              - Codebase
          - ⚠️ Every task your organization has to do repeatedly should be SCALABLE
            - "SCALABLE" = "sublinear scaling with regard to human interactions"
            - 👍 Policies are a wonderful tool for making process scalable
        - Policies That Don't Scale
          - 👎 A traditional approach to depreca-tion: "We'll delete the old Widget on August 15th; make sure you've converted to the new Widget."
            - 👍 "Churn Rule"
              - Infrastructure teams must do the work to move their internal users to new versions themselves(!) or do the update in place, in backward-compatible fashion
                - ❓ Q: e.g. you made the change of the shared library - you upgrade it's usage everywhere, right?
          - 👎 A traditional use of development branches => "We need tighter controls on when things merge. We should merge less frequently"
        - Policies That Scale Well
          - "If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn't surfaced by tests in our Continuous Integration (CI) system, it is not the fault of the infrastructu-re change."
            - 👍 "The Beyoncé Rule."
              - 🙂 "If you liked it then you shoulda put a ring on it."
              - 🔥 "If you liked it, you should have put a CI test on it"
          - 👍 Expertise and shared communica-tion forums offer great value as an organization scales
            - New experts grow
        - Example: Compiler Upgrade
          - A compiler upgrade will almost always result in minor changes to behavior
            - 👎 In Google the 2006 compiler upgrade was extremely painful
              - many Hyrum's Law problems
              - didn't have the Beyoncé Rule yet
          - 👍 The more frequently you change your infrastructure, the easier it becomes to do so.
            - Expertise
              - We know how to do this; for some languages, we've now done hundreds of compiler upgrades across many platforms
            - Stability
              - There is less change between releases because we adopt releases more regularly; for some languages, we're now deploying compiler upgrades every week or two.
            - Conformity
              - There is less code that hasn't been through an upgrade already, again because we are upgrading regularly.
            - Familiarity
              - Because we do this regularly enough, we can spot redundancies in the process of performing an upgrade and attempt to automate. This overlaps significantly with SRE views on toil
            - Policy
              - We have processes and policies like the Beyoncé Rule. The net effect of these processes is that upgrades remain feasible because infrastruc-ture teams do not need to worry about every unknown usage, only the ones that are visible in our CI systems.
        - Shifting Left
          - 👍 The idea that finding problems earlier in the developer workflow usually reduces costs
          - 

          - Timeline of the developer workflow
      - Trade-offs and Costs
        - ⚠️ Within a Google there is a strong distaste for "because I said so"
          - 👍 the goal is consensus, not unanimity
            - It's fine and expected to see some instances of "I don't agree with your metrics/valuation, but I see how you can come to that conclusion."
        - Should calculate costs when making decision between few options
          - Cost
            - Financial costs (e.g., money)
              - is usually not the limiting factor
            - Resource costs (e.g., CPU time)
            - Personnel costs (e.g., engineering effort)
              - Is usually a limiting factor
              - 👍 keeping engineers happy, focused, and engaged can easily dominate other factors
            - Transaction costs (e.g., what does it cost to take action?)
            - Opportunity costs (e.g., what does it cost to not take action?)
            - Societal costs (e.g., what impact will this choice have on society at large?)
          - Consider biases
            - Status quo bias
            - Loss aversion
            - Others...
        - Example: Markers
          - 👎 How often have you been in a meeting that was disrupted by lack of a working marker?
            - All for a product that costs less than a dollar
          - 👍 Google tends to have unlocked closets full of office supplies, including whiteboard markers, in most work areas
          - We often say, "Google is a data-driven culture."
            - ⚠️ even when there isn't data
              - there might still be evidence, precedent, and argument
          - Decisions are made
            - because we must (legal requirements, customer
            - because it is the best option (as determined by some appropriate decider) we can see at the time, based on current evidence.
            - 👎 should not be "We are doing this because I said so."
        - Inputs to Decision Making
          - 2 scenarios
            - All of the quantities involved are measurable or can at least be estimated
              - 👍 We should have conversion table: N CPU = M RAM = O network bandwidth = X engineering hours = Y support hours ....
            - Some of the quantities are subtle, or we don't know how to measure them
              - no easy answer
                - We rely on experience, leadership, and precedent to negotiate these issues
          - Example: Distributed Builds
            - 👎 From 60 to 70% of developers build locally
            - ⚠️ Consider cost of running build farm vs weeks/months of saved time
            - ❓ Q: are there tools for distributed builds for TypeScript? for Golang?
            - 👍 Google introduced distributed build system
              - 👎 But soon bloated or unnecessary dependencies in the build graph became all too common.
                - ⚠️ "Jevons Paradox"
                  - consumption of a resource may increase as a response to greater efficiency in its use
                - So - Google did not foresee all of the costs
                  - ⚠️ Even a relatively simple trade-off of the form "We'll spend $$$s for compute resources to recoup engineer time" had unforeseen downstream effects!
          - Example: Deciding Between Time and Scale
            - ⚠️ Conflict between time and scale: should we add a dependency or fork/reimplement it to better suit our local needs?
              - Forks are ok
                - If project life span is short
                - If scope is limited
              - Forks are not ok
                - for interfaces that could operate across time or project-time boundaries (data structures, serialization formats, networking protocols)
        - Revisiting Decisions, Making Mistakes
          - Data informing decisions - 👍
            - but the data will change over time
              - ⚠️ => decisions will need to be revisited from time to time
          - 👍 the deciders need to have the right to admit mistakes
            - 👍 leaders who admit mistakes are more respected, not less!
      - Software Engineering Versus Programming
        - ⚠️ for a project that will last only a few days you don't need
          - Integration tests
          - Continuous Deployment
          - Semantic Versioning
            - ❓ Q: what is it?
          - Dependency Management