# Software engineering at Google

Made by: Manuel Méndez, María Rodríguez, Héctor Triguero, Ana Castro.

**Problems at Google**

At Google, challenges are not about the *type* of problem but rather the *scale*. The features and products they develop are often based on pre-existing concepts (e.g., email existed before Gmail), so their primary focus is on how to scale these products effectively.

**Code Readability and Style Guides**

Google places a strong emphasis on code readability by enforcing style guides and utilizing tools to maintain consistency. While adhering to these standards can slow down the development process, it ultimately improves maintainability and collaboration. For example, in C++, Google discourages the use of "auto" to ensure clarity and explicitness in the code.

**Communication and Collaboration**

Google encourages openness in coding practices to facilitate early feedback and continuous improvement. Engineers are urged to engage in discussions, share ideas, and avoid the *genius myth*—the false notion that a single programmer must have all the answers. Instead, collaboration is key to building robust and scalable systems.

**Measuring Productivity**

To assess productivity, Google first consults engineering leaders to determine what aspects need improvement and what metrics would be meaningful. They may then choose to conduct measurements discreetly or inform engineers of the evaluation process without revealing specific details. This approach ensures that developers do not alter their behaviour simply to optimize for a metric. An example of this is Google's **Code Health Team**, which focuses on maintaining long-term code quality rather than short-term output.

**Testing at Google**

- Boilerplate tests: implement first the basic scenarios then move on to the borderline conditions. Tests give you a good feeling for how a user is going to use the system in the common case.
- Flaky tests: tests that behave differently or appear to behave differently under the same conditions. Expensive because we must run them multiple times to check if they are flaky and, if the people running them aren't on our team, we force them to learn the system and figure out why it failed. At Google, they rerun tests to try and determine if they are flaky. The continuous integration systems will determine if the flaky test is worth running or not and send a signal to the tester so they can decide if they should look at it or not.
- Test double: when testing against complex systems, it is useful to use "mocks" since the important thing for is testing our software. Substitutes/Stand by for a complex system that emulates the responses that they would give so we don't have to use the actual system.
- Test coverage: the number of lines of code we execute in our tests. Checks if we are evaluating enough paths of our code. They also use mutation.

**Static analysis**

Static analysis is a technique used to examine source code without compiling or executing it. By analysing the code early in the development cycle, engineers can detect potential issues before they manifest in runtime. While static analysis cannot catch every possible problem, it helps identify many errors efficiently, reducing the cost of debugging later in development.

**Tricorder: Google's Static Analysis Tool**

Tricorder is Google's scalable static analysis platform. Rather than being a standalone tool, Tricorder acts as a **framework** that integrates multiple language-specific analysis tools developed by domain experts. It aggregates and presents analysis results at relevant stages of development, such as during code review, ensuring that developers receive timely feedback on potential issues.

**Large-Scale Change (LSC)**

A **large-scale change (LSC)** refers to modifications in the codebase that are too extensive to be implemented atomically. These changes cannot be made in a single step. Instead of requiring each team to make these changes independently, a **centralized team** is responsible for applying them across the entire organization.

Testing large-scale changes in their entirety is impractical. To mitigate this, changes are **designed to be modular**, allowing them to be split into smaller, independently testable parts. These segments are validated through continuous integration (CI) pipelines and reviewed by engineers with **global ownership** of the codebase.

**Constraints in Scalability**

For a system to remain scalable, organizations must impose **constraints**. Allowing every developer to choose their own compiler or operating system would undermine uniformity and reduce the efficiency of large-scale development processes. By enforcing **strategic constraints**, Google enhances collaboration, maintainability, and scalability across teams. Well-designed constraints empower rather than restrict engineers, enabling efficient large-scale engineering.

**Time as a Factor in Engineering Decisions**

The expected **lifespan** of software significantly influences engineering decisions. Long-term projects require careful planning to ensure maintainability, while short-term solutions may prioritize speed over sustainability. Understanding the software's intended longevity helps teams make informed choices about architecture, dependencies, and development strategies.