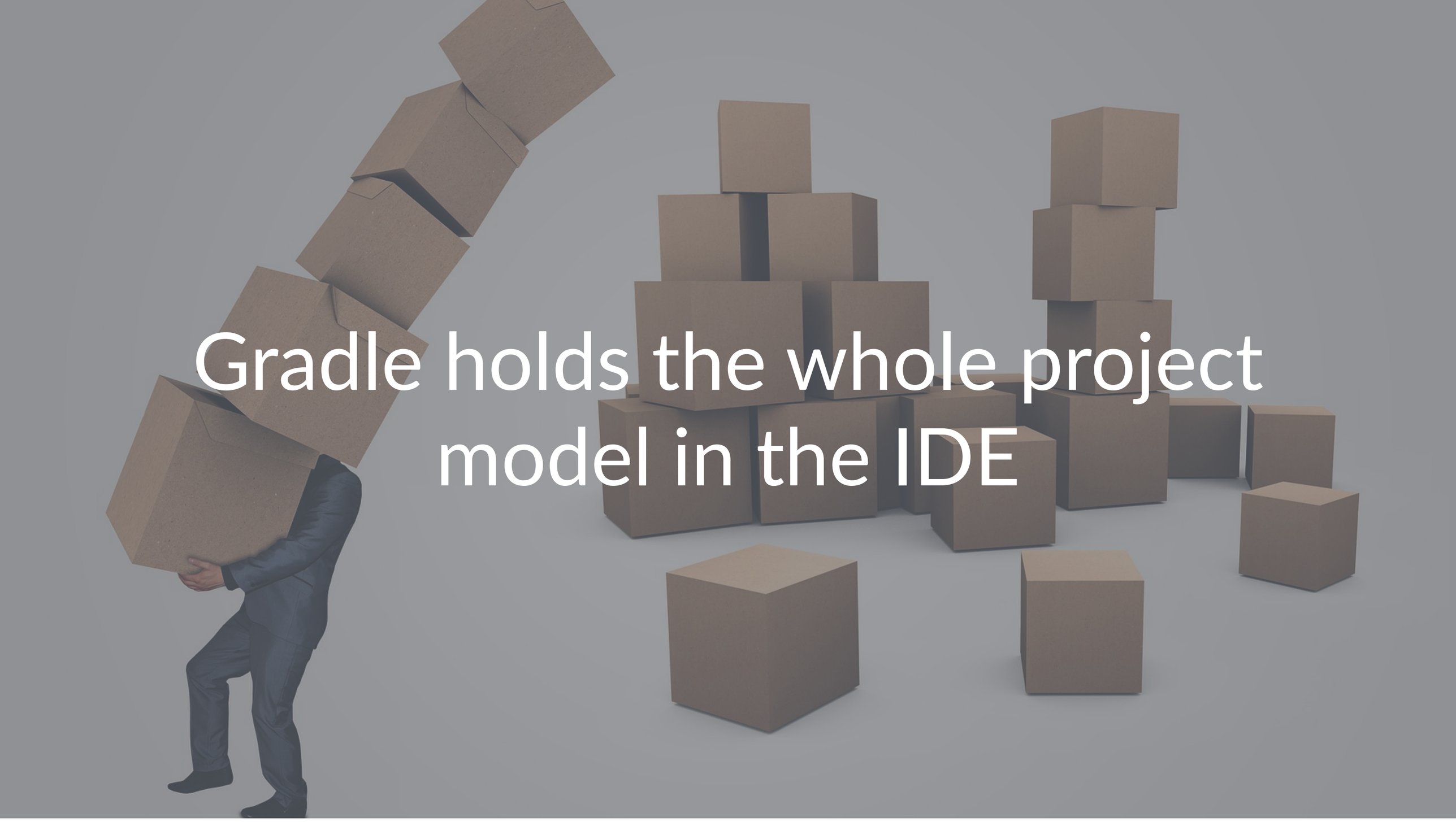
Working With Alternative Build Systems

Gautam Korlam & Kurt Nelson 🤒
Developer Platform at Uber

- Lets talk about the variois issues we see with Gradle builds at scale

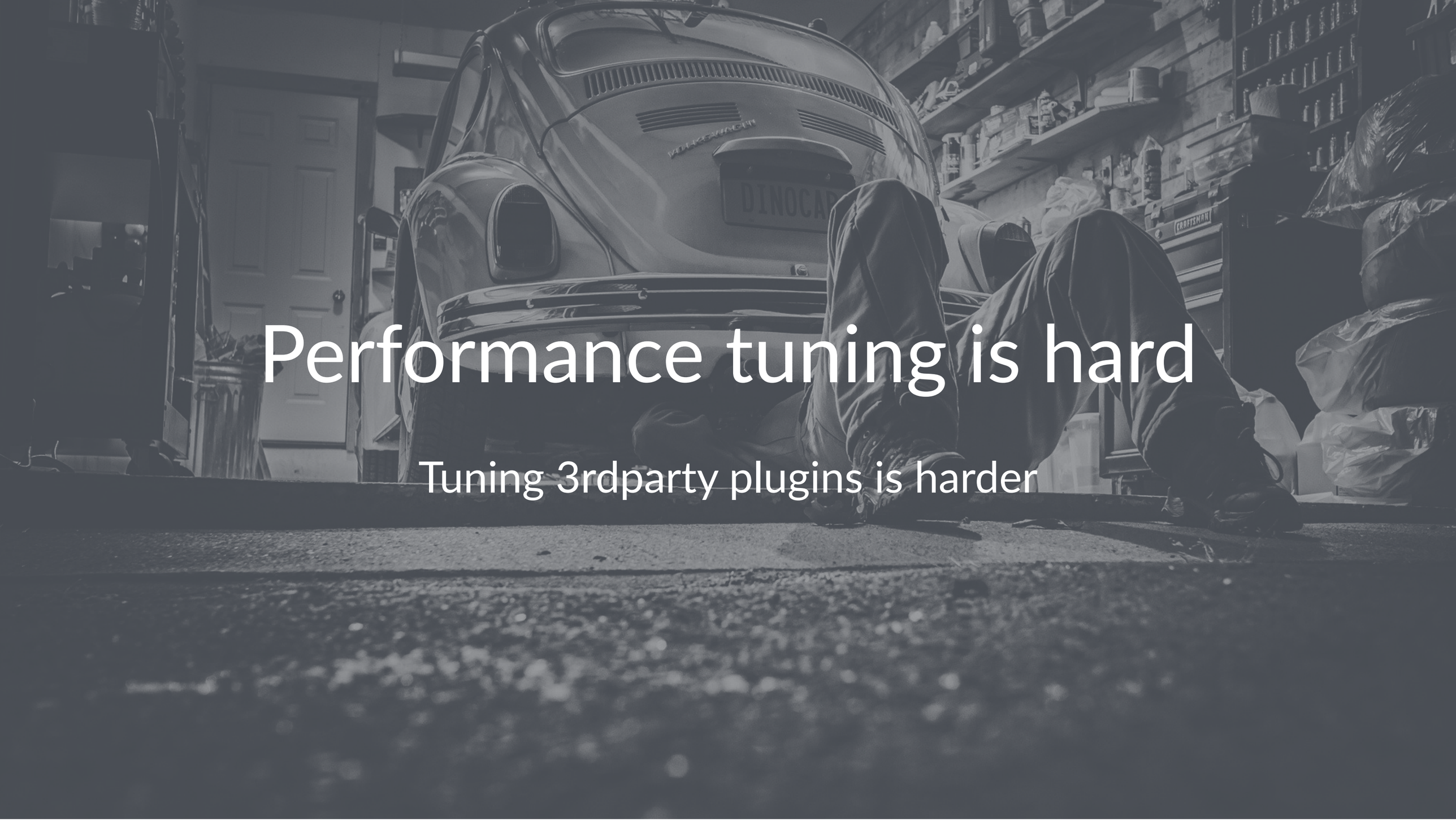
- The configuration phase is serial and not cacheable. Configuration is required anytime a dependency or build configuration changes
- Depends a lot on the various plugins in use as well
- At Uber, we had over 2000 modules that took 5+ minutes to configure
- Configuration on demand was not always reliable to stable builds

- With a full language, it is easy to write code with side effects
- A domain specific language or configuration language restricts functionality
- We had some scenarios where the timestamp or git sha would be part of the build signature
- There were other third party plugins that had side effects which caused builds to not be cached



Gradle holds the whole project model in the IDE

- Gradle stores a lot of info about the project model in memory
- As the number of modules increases the time and memory required to sync and hold the project model increases
- We have had several issues with memory leaks and resource hungry IDE issues over the years
- Gradle sync has to run when anything is changed, and can get very slow
- The user experience is not great as it is a non deterministic progress bar so we were never sure when it was going to actually finish



- Cost of knowledge for tuning performance is high and requires constant tuning as codebases grow
- More flavors/variants tend to make the task graph complex. There are limited levels that can be pulled to improve performance
- Extensibility is a double edged sword - Digging deeper into 3rdparty plugins to find issues is very time consuming
- Maintaining an ecosystem of plugins and making sure they all work with each other with compatibility is a lot of work

- As codebases, companies and engineering teams evolve, engineers working across tech stacks have high barriers for entry
- Decreased mobility of engineering resources makes development less efficient
- At Uber, we built the RIB architecture to be able to reason at an abstract level about product and feature development architecture
- It was hard however to use the same tools on both Android and iOS

- Remember the time when you could trust instant run builds?
- Reliable and reproducible builds are necessary to be able to diagnose issues and build trust
- Gradle tends to lag in this area

Dependency management

- Working with lots of dependencies requires good discipline to prevent problems like dependency fragmentation, unused and over dependency usage
- We had to define dependencies in a top level file and refer to them using constants
- Since developers dreaded having to run a gradle configuration all over again if there was a missing dependency, they would tend to add more than they needed, leading to less efficient build graphs

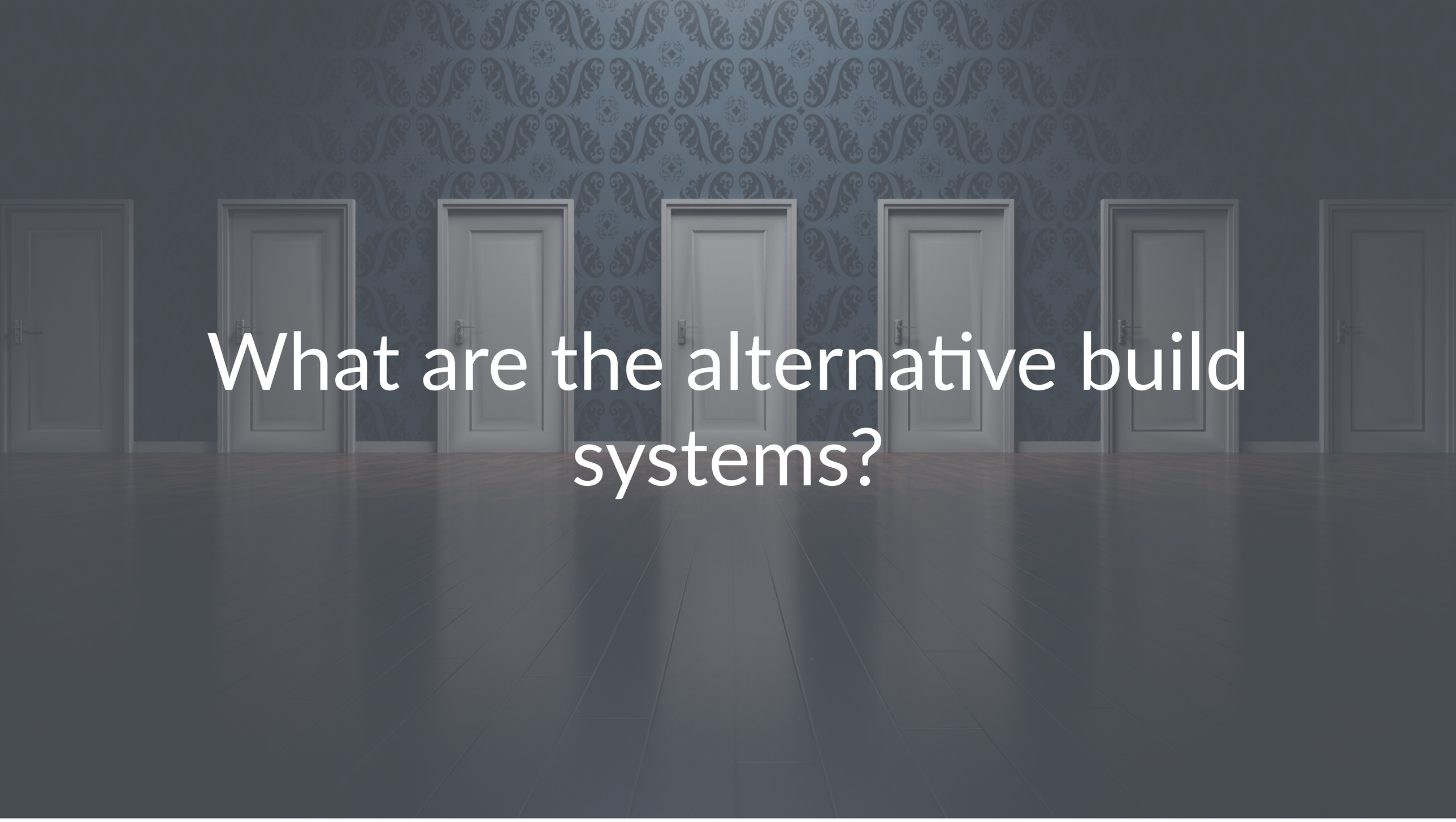
Extensibility with plugins

- Extensibility is available but is heavy weight
- Needing to understand plugin apis and being tied to the build system make it harder to migrate away or use standalone
- We had some internal plugins to perform common tasks like release, localization etc. The barrier to contribute to them was high enough that most devs would just work around problems than try to fix them as the investment was not worth it

- There is a good amount of feature set locked behind Gradle enterprise like the ability to introspect builds and stable caching infrastructure. Buying instead of building may not scale for all use cases
- Buying enterprise solutions might work for smaller/mid size companies with fewer engineering resources
- Larger companies would need much tighter integrations and capabilities


How do we fix these issues?

- Starlark (formerly known as Skylark) is a language intended for use as a configuration language
- It was designed for the Bazel build system, but may be useful for other projects as well
- Bazel uses Starlark as the notation for its BUILD files which declare the executables, libraries, and tests in a directory
- It is also used for its macro language, through which Bazel is extended with custom logic to support new languages and compilers



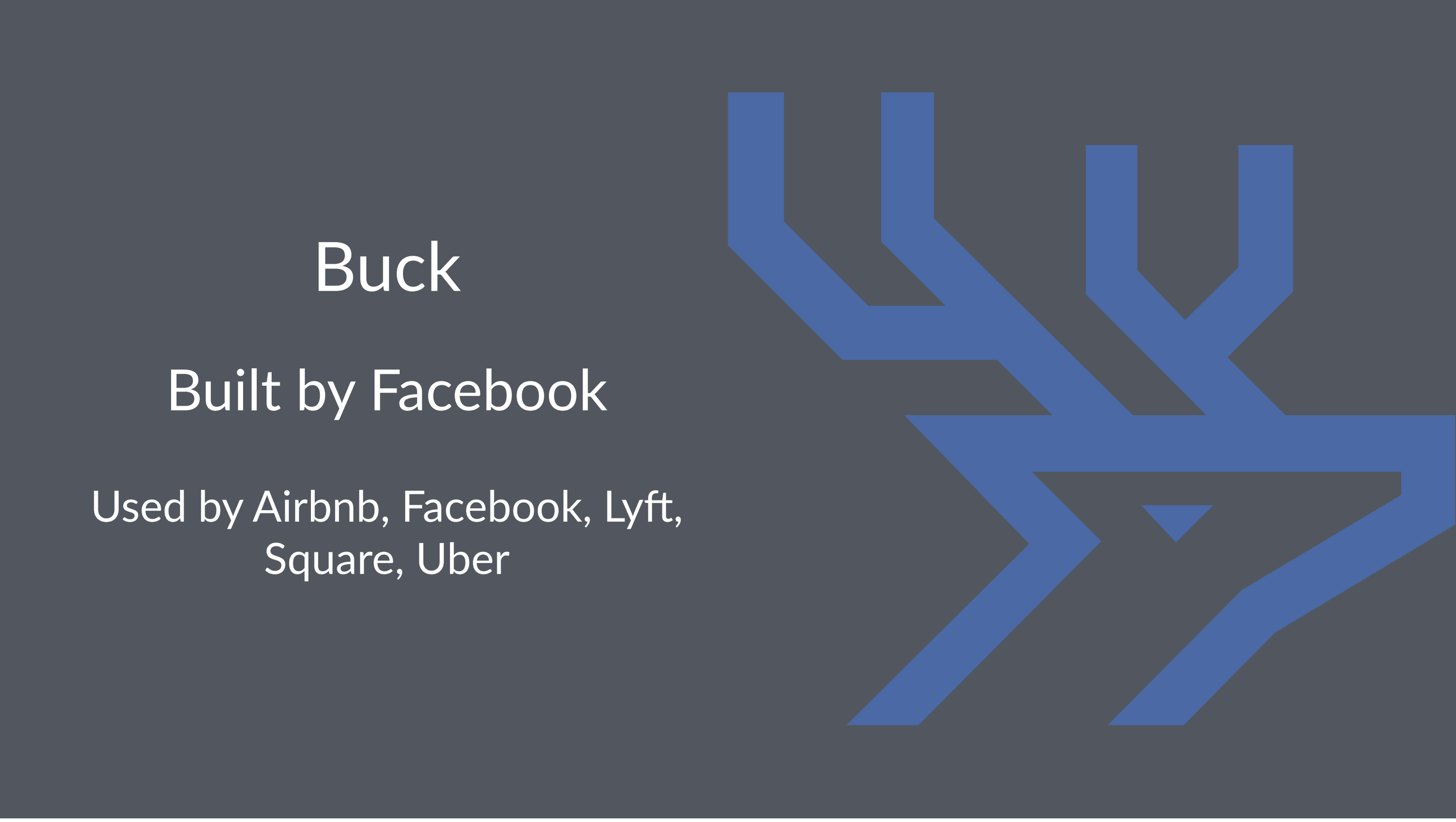
- Build systems that use starlark!

Bazel

Buck

Bazel and Buck are language and toolchain agnostic build systems that use starlark and are tailored towards large modular codebases

- Bazel is known for fast and reproducible builds It has a growing community and official backing from Google
- Google uses an internal fork of bazel knows as Blaze for almost everything
- Fast, Incremental, distributed, remote builds and fully extensible with custom rules similar to plugins in Gradle



- Very similar to Bazel in design due to being written by ex-Googlers who missed Blaze
- Was built originally focused on mobile development so has deeper optimizations for android and ios
- Fast, Incremental builds but distributed and remote builds require non trivial infra investment
- Extensibility is limited and requires modifications to core

Okbuck

Built by Uber

Hybrid inter-op between Gradle and Buck

- Built and used by Uber for Android and Java backend builds
- Hybrid solution to allow inter-op between Gradle and Buck as a transitory step
- Can be customized to allow bazel interop as well
- Requires heavy maintenance as gradle and AGP evolve
- Is not incremental so scales poorly with number of gradle modules
- Used by Airbnb, Lyft, Square, Uber
- We are in the process of moving to Bazel from Buck so the talk will focus on Bazel from now

Example configurations using Starlark and Bazel

Example android kotlin app[1]

```
load("@io_bazel_rules_kotlin//kotlin:kotlin.bzl", "kt_android_library")

kt_android_library(
    name = "lib",
    srcs = glob(["java/com/example/bazel/*.kt"]),
    custom_package = "com.example.bazel",
    manifest = "AndroidManifest.xml",
    resource_files = glob(["res/**"]),
    visibility = ["//:__pkg__"],
    deps = [
        "@maven//:androidx_appcompat_appcompat",
        "@maven//:androidx_core_core",
        "@maven//:androidx_core_core_ktx",
        "@maven//:androidx_drawerlayout_drawerlayout",
        "@maven//:androidx_fragment_fragment",
        "@maven//:androidx_lifecycle_lifecycle_common",
        "@maven//:androidx_lifecycle_lifecycle_viewmodel",
    ],
)
```

---

[1] For more examples, checkout github

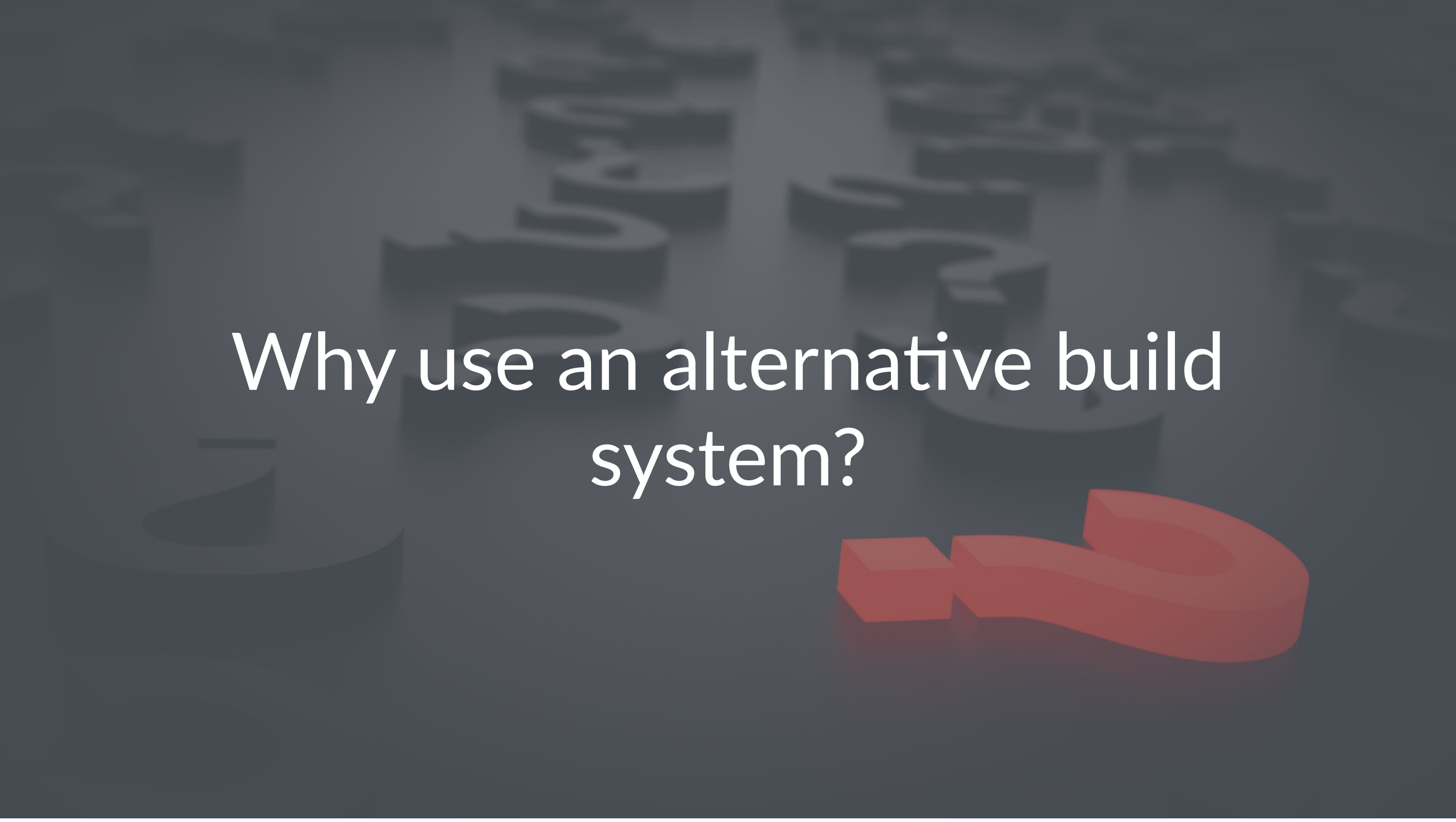
- Here we see an example starlark configuration in bazel to build an android app using kotlin sources

Example android kotlin app[1]

```
android_binary(
    name = "app",
    custom_package = "com.example.bazel",
    manifest = "AndroidManifest.xml",
    deps = [
        ":lib",
    ],
)
```

[1] For more examples, checkout github

– Here we see an example starlark configuration in bazel to build an android app using kotlin sources



Why use an alternative build system?

- Alternative build systems have advantages even at smaller scale

They are fast!

- Distributed build caches are reliable and in-memory
- Remote builds so machines are possible so developers are not constrained by laptop resources
- File system monitors to help the build system incrementally configure itself on local builds

Scale better as your codebase grows

- Fundamental design differences that optimize for lots of small focused modules
- Very low configuration overhead since the action/ target graphs can be cached

Design differences

Gradle	**Bazel**
Task Graph	Action Graph
Lazy if configured by task	All targets lazy by default
Combining different languages in one build is difficult	Fully native graph of different languages possible

Combining Languages

```
cc_library(
    name = "main-jni-lib",
    srcs = [
        "@local_jdk//:jni_header",
        "@local_jdk//:jni_md_header-linux",
        "Main.cc"
        ],
    hdrs = [ "Main.h" ],
    includes = [ "external/local_jdk/include", "external/local_jdk/include/linux" ],
)
```

```
cc_binary(
    name = "libmain-jni.so",
    deps = [ ":main-jni-lib" ],
    linkshared = 1,
)
```

Combining Languages

```
java_binary(
    name = "Main",
    srcs = [ "Main.java" ],
    main_class = "Main",
    data = [ ":libmain-jni.so" ],
    jvm_flags = [ "-Djava.library.path=." ],
)
```

Fool proof extensibility

Gradle	**Bazel**
Caching is opt-in	Caching is built in
Easy to have side effects	Impossible to have side effects
Need to understand plugin api and internals	Standard Starlark
Modifying existing behavior is not straight forward	All rules can be wrapped in a macro

Extending using macros[2]

```python
def kt_android_library(name, exports = [], visibility = None, **kwargs):
    """Creates an Android sandwich library. `srcs`, `deps`, `plugins` are routed to `kt_jvm_library` the other android
    related attributes are handled by the native `android_library` rule.
    """
    native.android_library(
        name = name,
        exports = exports + _kt_android_artifact(name, **kwargs),
        visibility = visibility,
        testonly = kwargs.get("testonly", default=0),
    )
```

[2] For more details on bazel kotlin rules, checkout github

- Using starlark, existing rules can be wrapped in macros to add additional functionality
- They can be modified to run completely different set of actions as well

Extending using macros[2]

```
def _kt_android_artifact(name, srcs = [], deps = [], plugins = [], **kwargs):
    """Delegates Android related build attributes to the native rules but uses the Kotlin builder to compile Java and
    Kotlin srcs. Returns a sequence of labels that wrapping macro should export.
    """
    base_name = name + "_base"
    kt_name = name + "_kt"

    base_deps = deps + ["@io_bazel_rules_kotlin//kotlin/internal/jvm:android_sdk"]

    native.android_library(
        name = base_name,
        visibility = ["//visibility:private"],
        exports = base_deps,
        **kwargs
    )
    _kt_jvm_library(
        name = kt_name,
        srcs = srcs,
        deps = base_deps + [base_name],
        plugins = plugins,
        testonly = kwargs.get("testonly", default=0),
        visibility = ["//visibility:private"],
    )
    return [base_name, kt_name]
```

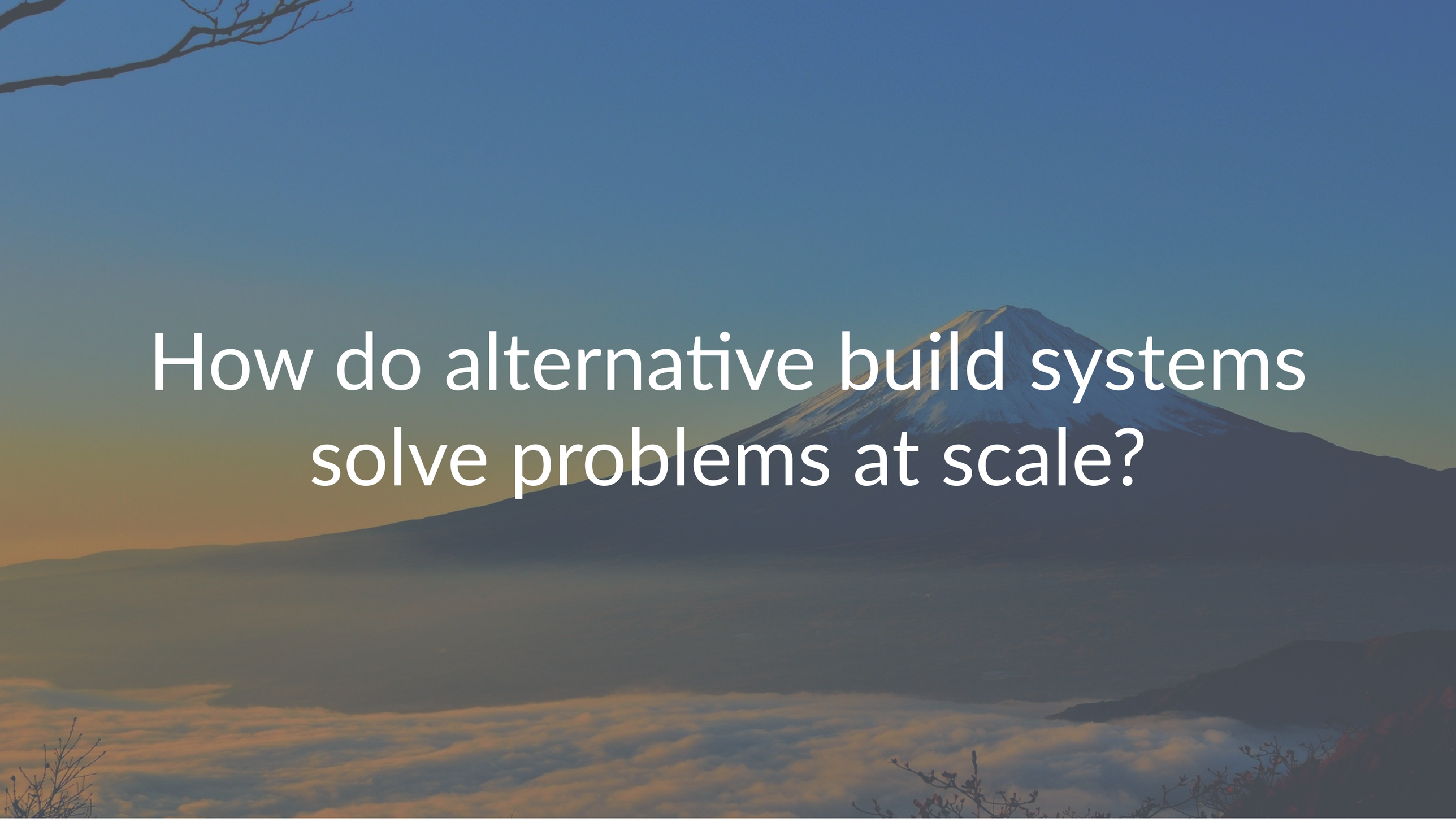
---

[2] For more details on bazel kotlin rules, checkout github

- Using starlark, existing rules can be wrapped in macros to add additional functionality
- They can be modified to run completely different set of actions as well

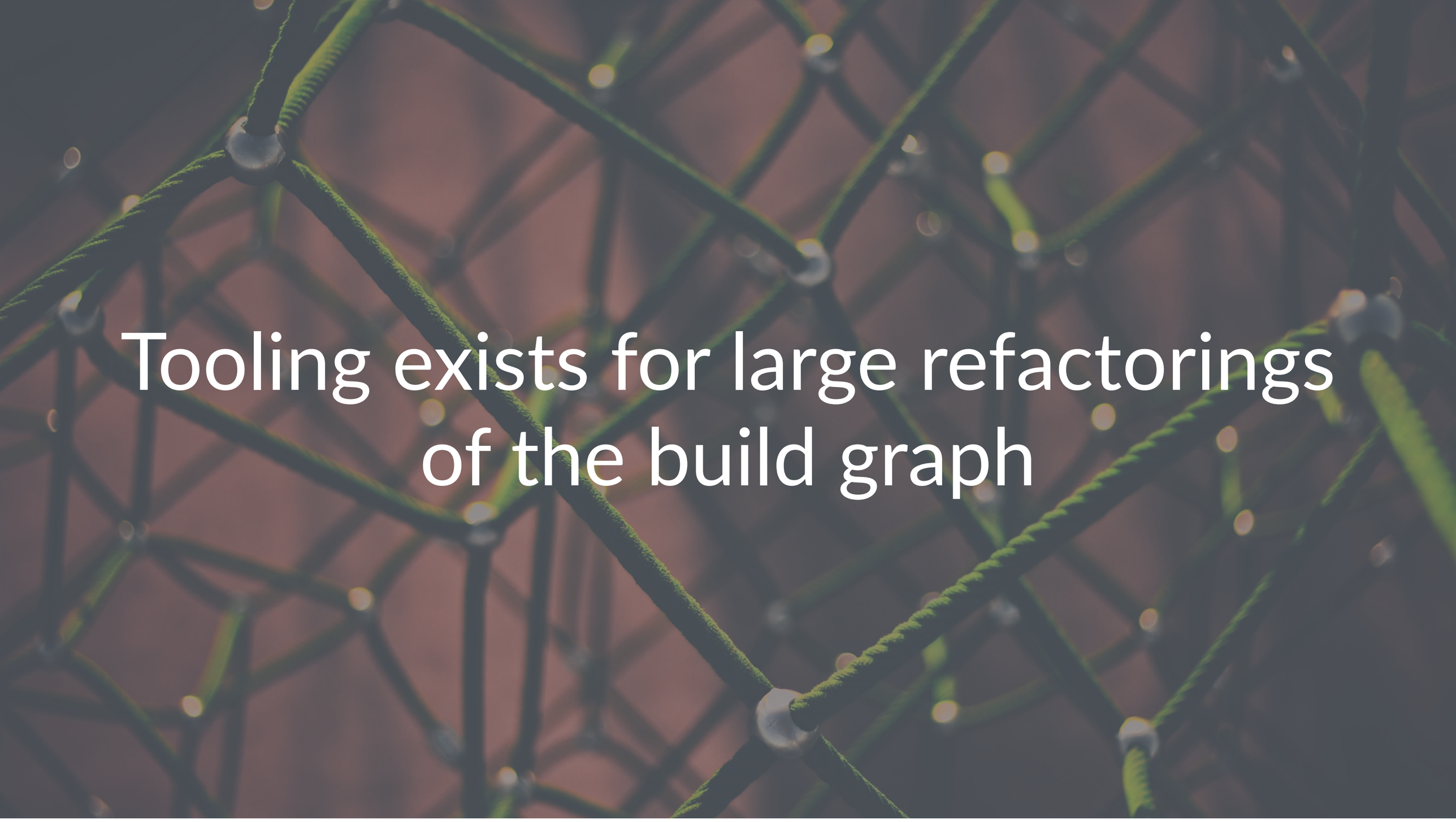
Organizational

Gradle	**Bazel**
Popular standard for Java/Kotlin	Any Language or platform, including iOS
Android developers know it already	Android developers can build iOS or backend
Follow Android plugin updates	Local control over when updates happen
Supported by Google and usable out of the box	Requires resourcing



- Now that we have talked about the differences, Let's take a look at how these alternative build systems tackle these challenges
- These systems begin to pay off more for the investment, the larger your codebase grows with time

- When there are a lot of builds even a 0.01% failure rate can be detrimental to developer productivity
- Bazel runs millions of builds every month on large codebases at various large companies
- Anecdote: It's used by spacex for building the software used on the rockets



- As a devx engineer that has to perform large codebase wide refactors, tooling to refactor BUILD configuration files is critical
- Bazel delivers with the buildozer tool
- Starlark not being a full blown programming language makes a big difference here
- We used to use find/replace with gradle files as the dsl parsers that exist are not very user friendly

Shared infrastructure for all

All languages are on equal footing

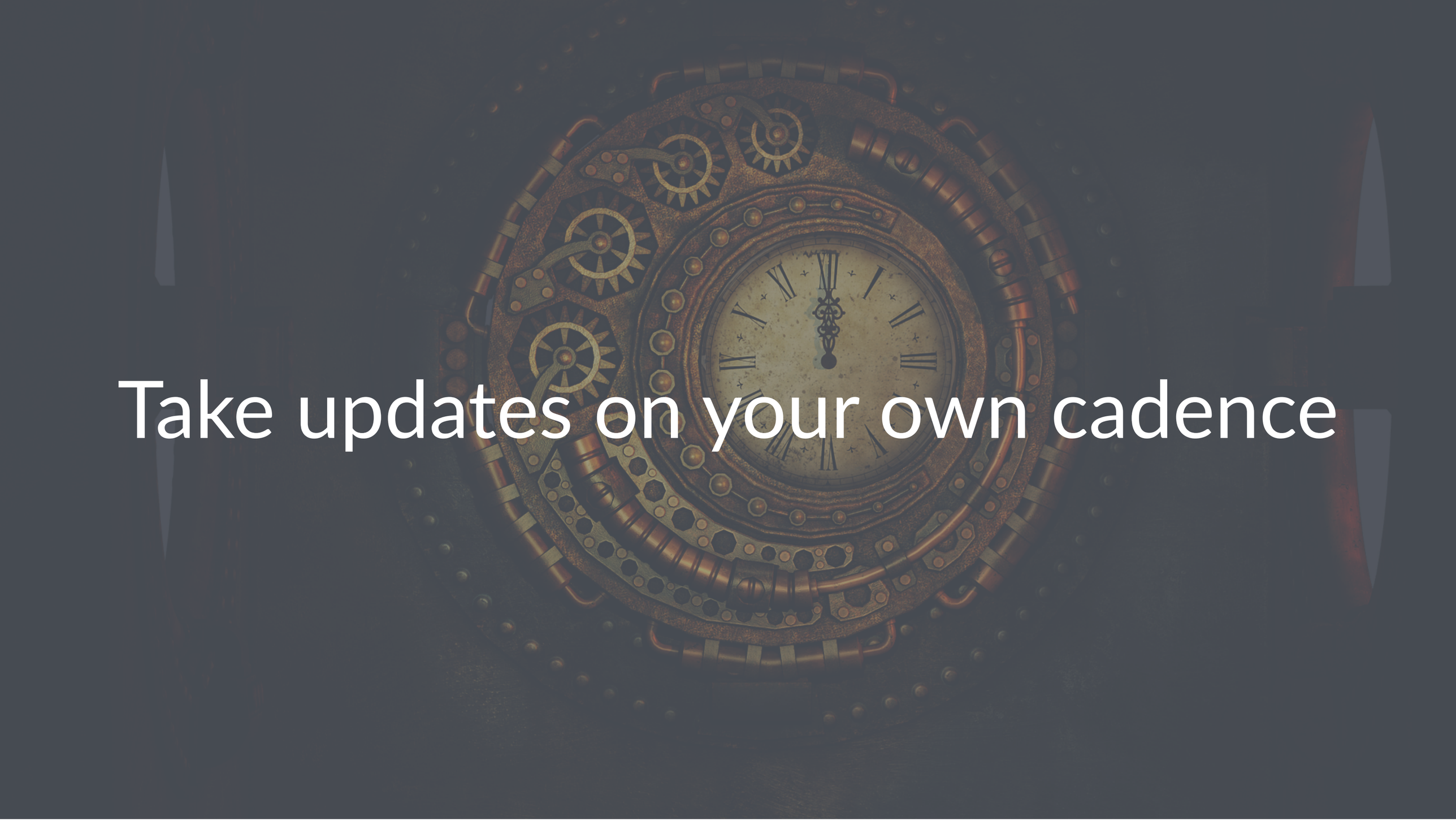
All targets can be remotely built and cached

All platforms can share common code like models

- We can spin up the cache and remote build infrastructure once for all code
- Maintenance is a lot easier
- An Android developer already knows how to compile and run tests for iOS code
- Backend engineers can validate that their changes don't break mobile builds

Changes the developer experience

- When there is a deadline and your build cache craps out or the gradle daemon dies unexpectedly on CI so you have to run a long resubmission to merge your code, you would wish for a more stable build system you can trust



- You become decoupled from the gradle release cycles, and can choose exactly what components you upgrade when
- For example, we were able to get D8 support before gradle did

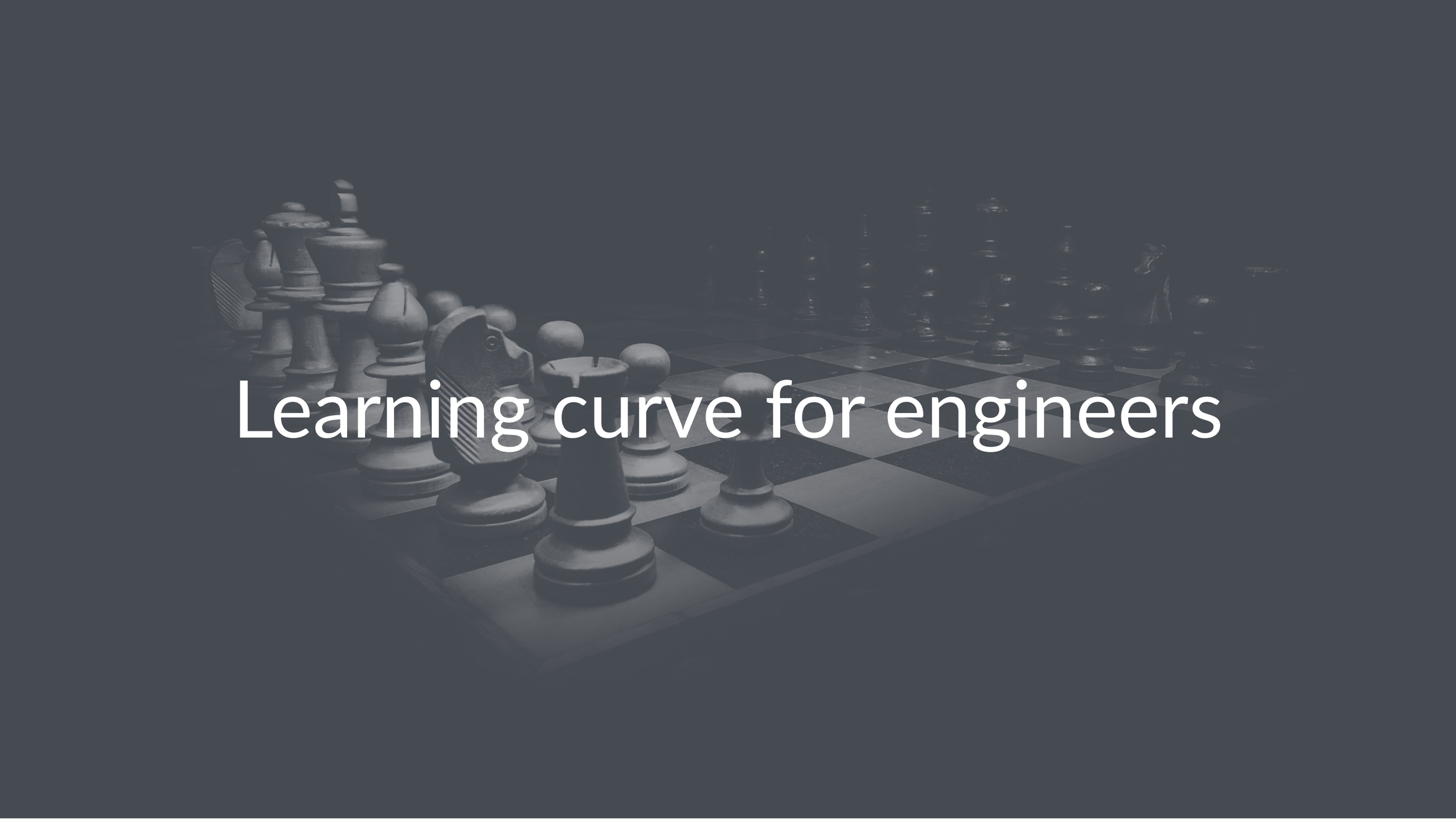
- Because of the way build targets are defined, it is easy to add a new target to the BUILD file without paying the configuration cost upfront
- There are also tools that help break down large targets
- This encourages developers to create small targets that build fast

- We saw cross team and org collaboration increase as there is an agency for everyone to contribute to improved tooling
- For example, we re-used a lot of common tooling across mobile and backend
- The barrier to entry of making changes in the build system is much lower due to safeguards around writing macros

- Since these are alternative systems, there are some caveats when using them as well
- But we believe the upsides outweigh them

- The latest shiny features are usually available with Gradle as the tools team at Google prioritizes for that first
- This has changed a lot lately with them not coupling everything tightly to Gradle however
- For example, we still don't have good support for dynamic app bundles

- With Buck, we use Intellij instead of Android Studio for greater compatibility, but that is slowly changing
- Bazel has specific plugins for Intellij and Android Studio
- Not all features work out of the box however
- For example, we still don't have layout preview at Uber

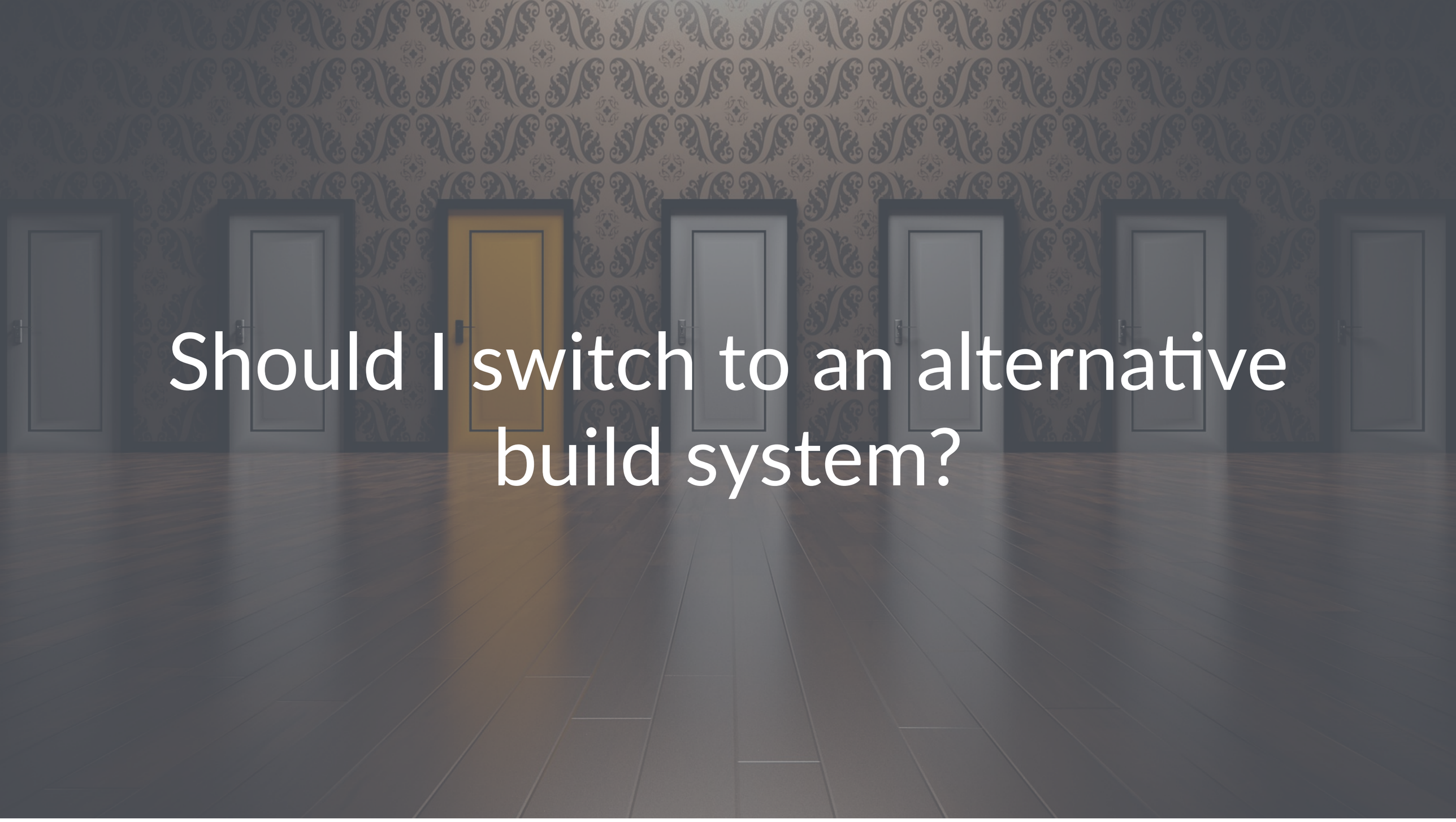


Learning curve for engineers

- As with any new system, developers need to get acquainted with how Bazel and Starlark work
- Developers have to write BUILD files instead of Gradle
- Authoring can be made easier using custom rules with sane defaults and macro support

[3] More info about BazelCon

- These systems are not as popular as gradle so there are not many community resources on stackoverflow, blog posts etc
- We have begun to see movement in the right direction so far and there is BazelCon happening every year as well
- This year it is happening in Sunnyvale in december



Should I switch to an alternative build system?

- Most teams can stay on gradle
- Gradle has been getting better about performance with recent releases as well
- You should evaluate your needs and look at the trade-offs we described so far to see if alternative systems work you
- If you want to learn a new system and need to build multiple languages seamlessly, they might be right for you

When should I consider it?

- When gradle is too slow for you
- On a more serious note, if you are looking to build large monorepos or want to have interop between multiple languages, that would be the right time to consider an alternative system
- We transitioned gradually as migration efforts can take a while
- Make sure you take into account the time required to migrate and onboard your engineers when you decide

How do I go about switching to an alternative?

- At Uber, we were able to use OkBuck as a go-between
- We used it to translate gradle files to buck for faster builds and eventually to move to Bazel
- You can switch from buck to bazel by using custom rules and buildozer to perform refactorings of build configurations

Thanks

Questions?