



# Developer-First Gradle

Verständliche Builds durch klare Trennung von  
Zweck und imperativer Logik



```
speaker {  
  name = "Stefan Wolf"  
  role = "Principal Software Engineer @ Gradle"  
  github = "wolfs"  
}
```

# Table of Contents

What is Gradle

Challenges

Vision & Principles

Building blocks

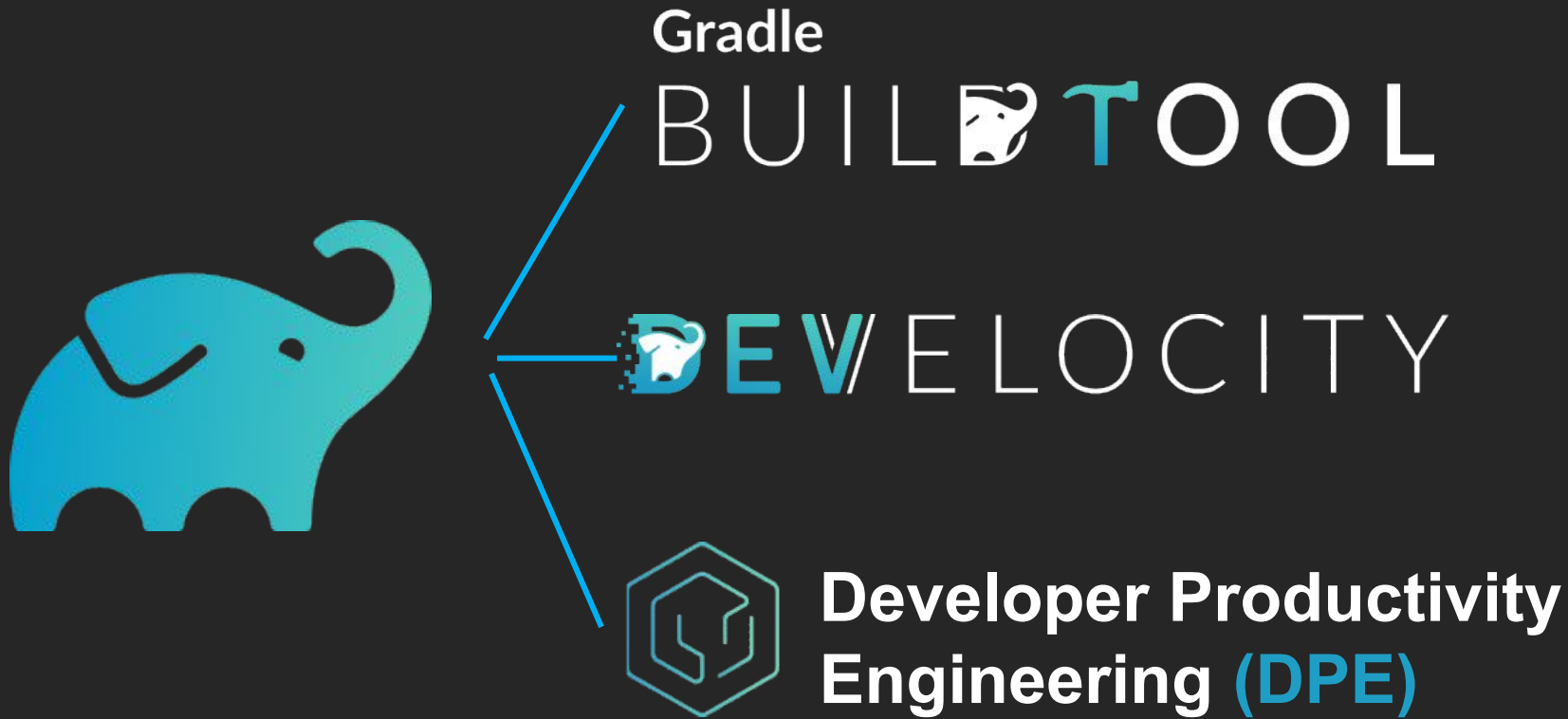
Declarative Gradle Early Access Preview 3

What now?

What's next in Declarative Gradle?



# What is Gradle, really?



# Gradle Build Tool accelerates developer productivity

Gradle is the open source build system of choice for Java, Android, and Kotlin developers. From mobile apps to microservices, from small startups to big enterprises, it helps teams deliver better software, faster.

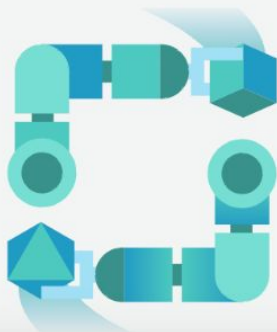
[Install Gradle 8.11.1](#)

[Get Started Guides](#)

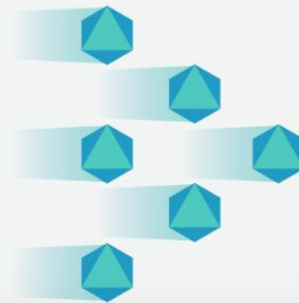
## Build Anything

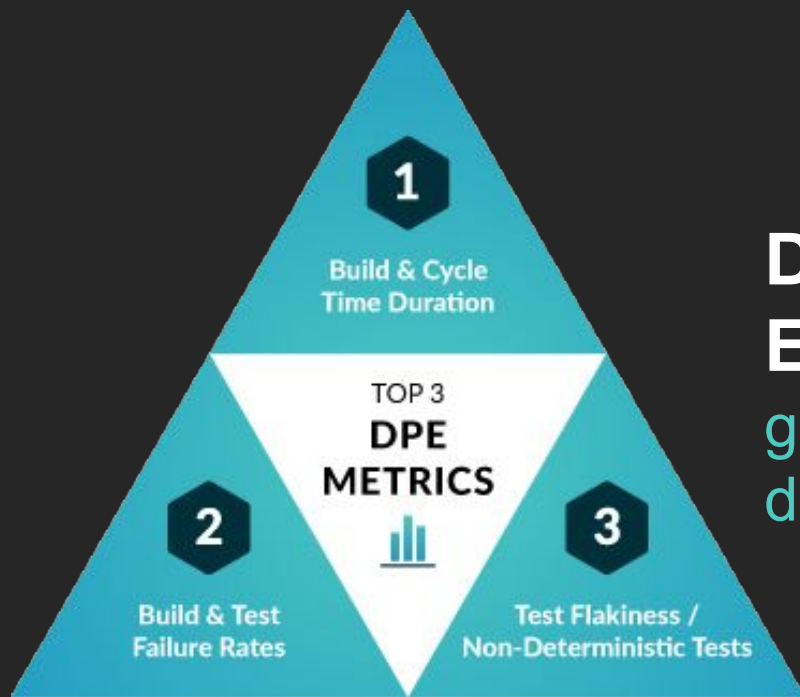


## Automate Everything



## Deliver Faster





# Developer Productivity Engineering (DPE)

[gradle.com/  
developer-productivity-engineering](https://gradle.com/developer-productivity-engineering)



# DEVELOLOCITY

(f.k.a. Gradle Enterprise)





# Faster, more reliable builds & tests

Tired of waiting on slow builds and tests? Need instant visibility to troubleshoot a failed build? Wish you could tell if that test is actually flaky? Develocity can help.



<https://gradle.com/develocity/>





**Achieve software  
delivery excellence  
in the age of AI**



# Ask the audience

Who knows Gradle?

Who uses Gradle Build Tool?

Who knows Develocity / [scans.gradle.com](https://scans.gradle.com)?

Who uses Develocity / [scans.gradle.com](https://scans.gradle.com)?



# Challenges



# Challenges

Gradle is flexible and extensible, but there are drawbacks...

- ❖ Build scripts are expressed in Gradle terms
- ❖ Build scripts can become complex
- ❖ Tooling/IDE can only help so much before it's just guessing



# Developer surveys say...

Gradle is used by a spectrum of roles.

- 🏠 Software Developers - Majority in most teams
  - Improve software by shipping features & fixing bugs
- 🏠 Build Engineers - Frequent in larger teams
  - Maintain the build and make Software Developers productive
- 🏠 ↔ 🧑 - Frequent in smaller teams
  - Who's the Gradle expert today?



# Ask the audience

Software Developer?  
Build Engineer?  
Both (at least 20% each)?



# Challenges

Software Developers need to think about Gradle plugins, tasks, and magic names to get their job done.

```
plugins {  
    java  
}  
repositories {  
    mavenCentral()  
}  
dependencies {  
    testImplementation(libs.junit.jupiter)  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
    api(libs.commons.math3)  
    implementation(libs.guava)  
}  
tasks.named<Test>("test") {  
    useJUnitPlatform()  
}
```



# Challenges

Complexity needs to be managed

```
plugins {  
    id("my-conventions")  
}  
apply { from("dependencies.gradle.kts") }  
tasks.named<Test>("test") {  
    useJUnitPlatform()  
    jvmArgs "-Dsamples=${projectDir.absolutePath}/samples"  
}
```

... 500 lines ...

```
tasks.named<Test>("test") {  
    useJUnitPlatform {  
        includeTags("Fast")  
    }  
}
```

# Challenges

How can automated tooling make sense of this?

```
android {  
    namespace = "com.example.${project.name}"  
}  
dependencies {  
    testImplementation(libs.junit.jupiter)  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
    if (!buildingForJava17()) {  
        implementation(libs.java17CompatibilityShim)  
    }  
    implementation(libs.guava)  
    listOf("foo", "bar").forEach { name ->  
        implementation("org:${name}:1.0")  
    }  
}  
fun buildingForJava17() = JavaVersion.current() == JavaVersion.VERSION_17
```

# Recommendations today

Gradle builds can look declarative to manage the challenges.

- Give your convention plugins meaningful names so Software Developers understand what they are
- Keep code in plugins to manage complexity
  - For custom build logic
  - For common defaults
- Keep your build scripts simple - condition and loop free



# Done?

This might not be enough.

```
plugins {  
    id("backend-library-conventions")  
}  
  
dependencies {  
    api(libs.commons.math3)  
    implementation(libs.guava)  
}
```






# Vision & Principles



# Vision

*Elegant and extensible declarative build language that allows developers to describe any kind of software in a clear and understandable way.*

- Extensible, flexible 
- Declarative 
- Clear and understandable 



# Needs of our users

To reliably build software, Gradle wants two things—the software definition and build logic.

- ◆ Software Definition - *What* needs to be built
  - Kind of software, languages, target platforms, dependencies
  - Lives in build files
  - Meant to be read and modified by Software Developers
- ◆ Build Logic - *How* the software will be built
  - Capabilities, integrations, organizational requirements
  - Lives in plugins
  - Meant to be read and modified by Build Engineers

# Goal of Declarative Gradle

Software Developers should not need to be Build Engineers too.

- Separate software definition and build logic with a declarative DSL
- Match the software definition to the software domain
- Excellent Tooling and IDE Integration





# Building blocks



# Declarative Gradle

Assembling the pieces...

- ❖ Convention plugins
- ❖ High-level models
- ❖ Kotlin DSL



# Ask the audience

How many convention plugins do you usually apply to a single project?

None?

Just one?

2-9?

10-50?

50+?



# Mixed concerns

Convention plugins today mix two different user concerns.

```
plugins {  
    id("java-library")  
}  
  
tasks {  
    val mySpecialTask by registering {  
        ...  
    }  
}  
  
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}  
  
dependencies {  
    implementation("org.apache.commons:commons-text:1.11.0")  
}
```

# Software Developer concerns

Software Developers care about some of this...

```
plugins {  
    id("java-library")  
}  
  
tasks {  
    val mySpecialTask by registering {  
        ...  
    }  
}  
  
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}  
  
dependencies {  
    implementation("org.apache.commons:commons-text:1.11.0")  
}
```

# Build Engineer concerns

Build Engineers care some of these things too.

```
plugins {  
    id("java-library")  
}  
  
tasks {  
    val mySpecialTask by registering {  
        ...  
    }  
}  
  
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}  
  
dependencies {  
    implementation("org.apache.commons:commons-text:1.11.0")  
}
```

# Shared location/Separate concerns

Encourages complexity. How could we split this?

- ❖ Software Developers need to wade through unfamiliar things
- ❖ Build Engineers need to maintain Software Developer things



# Software definition

What needs to be built?

- A software type is a high level model that a Software Developer can understand.
- For example, a JVM application is a software type.
  - A Software Developer knows they want to "make a JVM application".
  - A Software Developer knows the minimum version of the JVM they want to use.
  - A Software Developer knows which dependencies they want to use.
- A software type hides "low level" detail like tasks





# Software types

Software Developers care about 100% of this.

```
// in project definition
// NOTE: No other top-level blocks are allowed

javaLibrary {
    javaVersion = 17

    dependencies {
        implementation("org.apache.commons:commons-text:1.11.0")
    }
}
```



# Software types

Build Engineers care about under the hood...

- Software types replace applying plugins in a project definition file
  - There's a single software type for each project
- Think of these like project extensions
- Gradle still uses build logic and plugins to transform the software type into work to be done
- EAP3 has prototype plugins to demonstrate what this would look like (demo later)



# Reusable configuration

Configuration that is Software Developer facing can be reused without knowing about Gradle plugins.

```
// in settings
defaults {
    javaLibrary {
        javaVersion = 17

        dependencies {
            implementation("org.apache.commons:commons-text:1.11.0")
        }
    }
}
```

```
// in project definition
javaLibrary {
}
```



# Ask the audience

Which DSL do you use today to define your builds?

Kotlin DSL?

Groovy DSL?

Declarative Gradle (DCL)?



# Simplifying the software definition

Declarative Configuration Language (DCL)

- ❖ Text-based, human readable
- ❖ Document oriented
- ❖ Highly toolable

```
javaLibrary {  
    javaVersion = 17  
    dependencies {  
        implementation("...")  
    }  
}
```



# Language highlights

DCL is a strict, tiny subset of Kotlin

- ❖ Purely declarative
  - Simple assignments and nested blocks
- ❖ Forbids "code" constructs in definition files
  - No imports, loops, functions, conditional statements
- ❖ Fast and resilient parser
  - Parses even when there are errors



# Language highlights (cont)

Borrows many patterns/ideas from existing DSLs

- ◆ Supports simple property assignments
- ◆ Supports creating elements in containers
- ◆ Supports lists and file references

```
javaApplication {  
    javaVersion = 17  
    jvmArguments += listOf("-Xmx64m")  
    checkstyle {  
        configFile = layout.settingsDirectory.file("config/checkstyle.conf")  
    }  
}
```



# Tooling

Tooling (like IDEs) can do some interesting things

- ❖ Request project definitions without fully configuring the build
- ❖ Validate project definition against schema
- ❖ Request mutations to be made programmatically to the build definition





# Early Access Preview 3



# Disclaimer

These are all experiments and constantly changing/breaking.

- Sample projects require Gradle milestones and unreleased features
- Provided prototype plugins of Software Types are changing all the time and are not ready for production use.
- IDE features require Android Studio or IntelliJ nightlies



# IDE & Tooling Demo



# Demo - Recap

- ◆ IDE support for DCL in IntelliJ IDEA
  - Also works in Android Studio and Visual Studio Code
- ◆ Code completion is fast and concise



# Ask the audience

Which IDEs do you use most of the time?

IntelliJ IDEA?

Android Studio?

Visual Studio Code?

Eclipse?

Apache NetBeans?

Other?



# Limitations as of now

- ◆ No composability/extensibility for Software Types
  - Requires writing a new software type
    - Adding extra functionality to an existing software type
    - Applying a third party plugin

```
javaLibrary {  
    // javaVersion is a built-in property of javaLibrary  
    javaVersion = 17  
  
    // checkstyle is an extension to javaLibrary  
    checkstyle {  
        toolVersion = "9.23"  
    }  
}
```

# Limitations as of now (cont)

- ◆ Missing support for model types
  - No `FileCollection`
    - Use `ListProperty<RegularFile>` or `ListProperty<Directory>` instead
  - No Maps
    - Use `NamedDomainObjectContainer<T>` with simple objects instead
  - No `PolymorphicDomainObjectContainer<T>`
    - No replacement for this yet



# Implementation Demo





`public interface CustomDesktopComposeApplication {` 11 usages  Tom Tresansky

`@Restricted` 1 usage  Tom Tresansky


`Property<String> getGroup();`

`@Restricted` 2 usages  Tom Tresansky

`Property<String> getVersion();`


`@Nested` 3 usages  Tom Tresansky


`KmpApplication getKotlinApplication();`

`@Configuring` no usages  Tom Tresansky

> `default void kotlinApplication(Action<? super KmpApplication> action) {` action.execute(getKotlinApplication  
> `()); }`

`@Nested`  Tom Tresansky

 `Compose getCompose();`

`@Configuring`  Tom Tresansky

> `default void compose(Action<? super Compose> action) {` action.execute(getCompose()); }

`@Nested` 3 usages  Tom Tresansky

`SqlDelight getSqlDelight();`

```

/**
 * Universal APIs that are available for all {@code dependencies} blocks.
 *
 * @apiNote This interface is intended to be used to mix-in DSL methods for {@code dependencies} blocks.
 * @implSpec The default implementation of all methods should not be overridden.
 * @implNote Changes to this interface may require changes to the
 * {@link org.gradle.api.internal.artifacts.dsl.dependencies.DependenciesExtensionModule} extension module for Groovy DSL or
 * {@link org.gradle.kotlin.dsl.DependenciesExtensions} extension functions for Kotlin DSL.
 *
 * @see <a href="https://docs.gradle.org/current/userguide/implementing_gradle_plugins_binary.html#custom_dependencies_blocks">Creating custom
dependencies blocks.</a>
 *
 * @since 7.6
 */

```

```

@SuppressWarnings("JavadocReference") 吕 Sterling Greene +2

```

```

public interface Dependencies {
    /**
     * A dependency factory is used to convert supported dependency notations into {@link org.gradle.api.artifacts.Dependency} instances.
     *
     * @return a dependency factory
     * @implSpec Do not implement this method. Gradle generates the implementation automatically.
     */
}

```

Basic types of dependencies used by either an application or library, for production or test code.

```



@SuppressWarnings("UnstableApiUsage")

```

```

public interface BasicDependencies extends Dependencies, PlatformDependencyModifiers {
    DependencyCollector getImplementation();
    DependencyCollector getRuntimeOnly();
    DependencyCollector getCompileOnly();
}

```

```
public interface SqlDelight { 3 usages  Tom Tresansky  
    NamedDomainObjectContainer<Database> getDatabases(); 2 usages  Tom Tresansky  
}
```

```
sqlDelight {  
    databases {  
        database("ApplicationDatabase") {  
            packageName = "org.gradle.client.core.database.sqldelight.generated"  
            verifyDefinitions = true  
            verifyMigrations = true  
            deriveSchemaFromMigrations = true  
            generateAsync = false  
        }  
    }  
}
```

# Implementation Recap

- ◆ Main principles
  - Reuse prototypes when modules are simple
  - Write Software custom types
- ◆ Feasibility
  - If you can reuse our prototypes plugin, it's easy
  - If you can't but are used to write Gradle Plugins, it's not that hard
- ◆ Documentation
  - Guide <https://declarative.gradle.org/docs/reference/migration-guide/>
  - Case Study <https://declarative.gradle.org/docs/reference/migration-case-study/>



# Ask the audience

How many software types do you think you'd have in your build?



# What now?



# Declarative build files

Without using Declarative Gradle

- Create software type convention plugins
  - add a top-level extension
  - map that configuration to “usual” plugins
  - use Kotlin DSL



```
plugins {  
    id("org.gradle.client.softwaretype.desktop-compose-application")  
}
```

```
desktopComposeApp {  
    group = "org.gradle.client"  
  
    // Version must be strictly x.y.z and  $\geq 1.0.0$   
    // for native packaging to work across platforms  
    version = "1.1.3"
```

```
    kotlinApplication {  
        dependencies {  
            implementation(platform("org.jetbrains.kotlin:kotlin-bom:2.0.21"))  
            implementation(platform("org.jetbrains.kotlinx:kotlinx-coroutines-bom:1.8.1"))
```

`@SuppressWarnings("UnstableApiUsage")` no usages  Tom Tresansky +1

```
public abstract class CustomDesktopComposeApplicationPlugin implements Plugin<Project> {  
    public static final String DESKTOP_COMPOSE_APP = "desktopComposeApp"; 1 usage
```

`@Override`  Tom Tresansky +1

```
public void apply(Project project) {  
    var projectDefinition = project.getObjects().newInstance(CustomDesktopComposeApplication.class);  
    project.getExtensions().add(CustomDesktopComposeApplication.class, DESKTOP_COMPOSE_APP,  
        projectDefinition);  
  
    wireKMPApplication(project, projectDefinition.getKotlinApplication());  
    project.getPluginManager().apply( pluginId: "org.jetbrains.kotlin.plugin.serialization");
```



```
sqlDelight {  
    databases {  
        create( name = "ApplicationDatabase") {  
            packageName = "org.gradle.client.core.database.sqldelight.generated"  
            verifyDefinitions = true  
            verifyMigrations = true  
            deriveSchemaFromMigrations = true  
            generateAsync = false  
        }  
    }  
}
```

```
sqlDelight {  
    databases {  
        database("ApplicationDatabase") {  
            packageName = "org.gradle.client.core.database.sqldelight.generated"  
            verifyDefinitions = true  
            verifyMigrations = true  
            deriveSchemaFromMigrations = true  
            generateAsync = false  
        }  
    }  
}
```



# Declarative build files

Without using Declarative Gradle

- ❖ No imperative logic in build files (not enforced)
- ❖ Only requires one convention (aka software type) plugin applied
- ❖ Only one top level block



# What's next?



# Early Access Preview 3 out now

<https://blog.gradle.org/declarative-gradle-april-2025-update>

- 🛡 This post has videos highlighting some of the same features you've seen in the demos today.
- 🛡 Declarative Gradle can be used by early adopters for simple projects
- 🛡 Add support for testing to our prototype plugins
- 🛡 More DCL features to support the official Android Software Type
  - File and directory properties
  - List properties
- 🛡 Discovery work on the migration of existing builds



# Challenges



Gradle is flexible and extensible, but there were ~~are~~ drawbacks...

- ❖ ~~Build scripts are expressed in Gradle terms~~
  - Software types are simpler and in the domain of the Software Developer
- ❖ ~~Build scripts can become complex~~
  - DCL files cannot contain code or do things in multiple ways
- ❖ ~~Tooling/IDE can only help so much before it's just guessing~~
  - Automated tooling can programmatically edit/understand build definitions



# Ask the audience

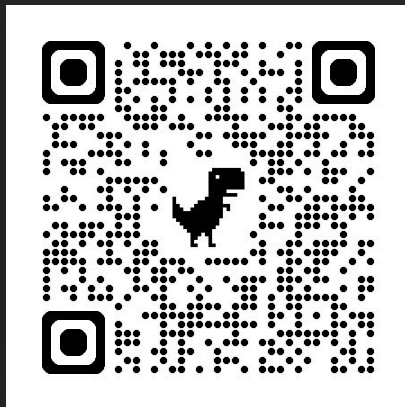
Have you tried Declarative Gradle already?



# Call-to-action: Give us feedback on EAP3

Submit feedback to <https://declarative.gradle.org/docs/feedback/>

- Try out a sample, create a new build or even try migrating a build.
  - This is a chance to influence a new Gradle feature very early.
- We're particularly interested in feedback from Software Developers who do not know Gradle well.
- You can also contact us on Slack and GitHub



# Roadmap

We are starting up on EAP4. <https://declarative.gradle.org/docs/ROADMAP/>

- ❖ Composability and extensibility prototypes
- ❖ Migration to declarative Gradle
- ❖ Backwards compatibility concerns
- ❖ Expected Q3 2025





# Roadmap (cont)

When will this be incubating? <https://declarative.gradle.org/docs/ROADMAP/>

- It depends...
  - On your feedback for EAP3 and EAP4
  - On making existing built-in plugins compatible
- Would like to make this happen in 2025
- Stable/GA would come later
  - Requires larger community feedback and adoption



# Team effort

We work on this together 🤝

<https://kotlinfoundation.org/news/building-better-developer-experience/>

- 🏠 Multiple teams at Gradle (DSL, Software, IDE)
- 🏠 Android Studio team at Google
- 🏠 IntelliJ & Kotlin teams at JetBrains



# Thank you!



```
speaker {  
  name = "Stefan Wolf"  
  role = "Principal Software Engineer @ Gradle"  
  github = "wolfs"  
}
```



[gradle.com/trial](https://gradle.com/trial)

## Request a Guided Develocity Trial

- 01 Install and configure
- 02 Connect your builds
- 03 Capture your current build data
- 04 Optimize your build
- 05 Quantify improvements
- 06 Present final report