Performance

Feature	Gradle 3.4.1	Maven 3.3.3
Incremental Builds	Gradle checks in between build runs whether the input, output or an implementation of a task has changed since the last build invocation. If not, the task is considered up to date and is not executed. Gradle also considers the configuration of the task as part of its input. So when you change the source compatibility setting for the compiler task it is no loner considerd up to date for example. The average build time is dramatically reduced for many builds by this feature.	
Incremental Compile for Java	Whether a the source code or the classpath changes, Gradle detects all classes that are affected by the change and will only recompile those.	

Feature	Gradle 3.4.1	Maven 3.3.3
Compile Avoidance for Java	If a dependent project has changed in an ABI-compatible way (only its private API has changed), then Java compilation tasks will be up-to-date. This means that if project A depends on project B and a class in B is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile A.	
API for Incremental Subtasks	When Gradle discovers that the input or output of a task has changed between build runs, the task is executed again. The task can use the incremental API to learn what files exactly have changed. With this information the task may not need to rebuild everything.	
Compiler Daemon		

Feature	Gradle 3.4.1	Maven 3.3.3
	When you need to fork the compilation process, Gradle creates a daemon process that is reused within a multi project build. This provides a dramatic speed improvement for the compilation process.	
Parallel Subproject Builds	In multi-module builds separate subprojects can be built in parallel.	

Dependency Management

Feature	Gradle 3.4.1	Maven 3.3.3
Transitive Dependencies	One of the main benefits of using a dependency management system is managing transitive dependencies. Gradle takes care of downloading and managing transitive dependencies.	One reason Maven 2 got popular is because of the support of transitive dependenciy management, a concept that was introduced into the Java world by Ivy in 2004.
API and implementation dependencies		

Feature	Gradle 3.4.1	Maven 3.3.3
	If a dependent project has changed in an ABI-compatible way (only its private API has changed), then Java compilation tasks will be up-to-date. This means that if project A depends on project B and a class in B is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile A.	
3rd Party Dependency Cache	Dependencies from remote repositories are downloaded and cached locally. Subsequent builds use cached artifacts to avoid unnecessary network traffic.	
Concurrency Safe Cache	Gradle's local dependency cache was designed from the ground up with concurrent saftey in mind. Whether it's utilizing Gradle's built-in parallel build support or running multiple Gradle	With Maven you can't run multiple builds at the same time against the same instance of the cache. There is an external Maven extension that provides this functionality. We don't know

Feature	Gradle 3.4.1	Maven 3.3.3
	builds at once, you can be sure your builds will remain fast and reliable.	how production ready this is and why it is not in the core, as it has been around a while.
Repository Aware Cache	Repository metadata is kept alongside cached dependencies. This allows Gradle to effectively handle cases in which artifacts differ for the same dependency in different repositories. Additionally, Gradle enforces that a cached dependency was retrieved from one of the project's configured repositories. If a dependency isn't available from an "approved" repository, the build will fail.	When different projects assigned to different repositories use the same cache they can overwrite each other when using the same dependency coordinates. This can lead to nasty, non-reproducible behavior, specially when dealing with dynamic dependencies.
Checksum Based Cache	Checksums are stored and used to both ensure cache integrity and optimize bandwidth usage. Artifacts are only downloaded if an existing cached version doesn't exist or remote	

Feature	Gradle 3.4.1	Maven 3.3.3
	version’s checksum is different.	
Sync Cache with Repository	Gradle has a --refresh-dependencies option to ignore all cached entries for resolved modules and artifacts. A fresh resolve will be performed against all configured repositories, with dynamic versions recalculated, modules refreshed, and artifacts downloaded. However, where possible Gradle will check if the previously downloaded artifacts are valid before downloading again. This is done by comparing published SHA1 values in the repository with the SHA1 values for existing downloaded artifacts.	
Parallel Download of Dependencies		When a dependency is not found in the cache it is downloaded from the remote repository. When multiple dependencies need

Feature	Gradle 3.4.1	Maven 3.3.3
		to be downloads, Maven downloads them in parallel.
Reads POM Metadata Format	Gradle is compatible with the POM Metadata format and can retrieve dependencies from any Maven compatible repository.	
Reads IVY Metadata Format	Gradle is compatible with the Ivy Metadata format. This includes declaring dependencies on particular configurations. Ivy metadata is exposed to custom resolution rules allowing you to filter on artifact branch, status or other custom metadata information.	
Dynamic Dependencies	Resolved dependency versions can be dynamic. Gradle supports the Maven snapshot mechanism but is more powerful than that. You can declare a dependency on the latest	Via Snapshot Dependencies

Feature	Gradle 3.4.1	Maven 3.3.3
	release, most current development version, or even the latest 4.x build.	
Dynamic Dependencies Selection Rules	Define custom rules to select a specific version when a dynamic dependency is declared. The rules can be based on names and version but also extended metadata like branch or status. The rules can also differ based on the environment the build is happening, e.g. local or CI.	
Version Conflict Resolution	By default, Gradle resolves conflicts to the newest requested version. You can customize this behavior.	Maven is not doing full conflict resolution. It simply always choses the version that has the lowest nesting level in the transitive dependency graph.
Central Versioning Definition	Declare library versions in a central location, avoiding duplication in your build	

Feature	Gradle 3.4.1	Maven 3.3.3
	scripts and establishing project conventions.	
Enforcing Central Versioning	While Maven allows projects to use dependency versions suggested by their parents, these version can be overridden. Gradle allows stricter governance of a dependency by using substitution rules to optionally force projects to use a particular version of a dependency.	
Substitution of Compatible Libraries	Use dependency substitution rules to identify that dependency should be treated as similar. For example log4j and log4j-over-slf4j. Tell Gradle that only one should be selected and use Gradle conflict resolution to pick the newest version from both of them. Similar use cases are situations where you have libraries like spring-all and spring-core in dependency	

Feature	Gradle 3.4.1	Maven 3.3.3
	graph. Without properly modelling this the proper behavior of your application depends on the very fragile order in your classpath.	
Modeling Releasable Units	Unique module dependencies are often logically associated to each other. In other words, a collection of dependencies should be treated like a single “releasable unit”. Think of a large application framework with many subcomponents. With Gradle, we can ensure we use the same version across dependencies, including transitive dependencies, to maintain reliable runtime compatibility.	
ReplacedBy Rules	Often projects migrate to new module names as part of a version upgrade. For example, Google’s “guava” project was formally known as “google-collections”.	

Feature	Gradle 3.4.1	Maven 3.3.3
	Declare that dependencies, including transitive dependencies, still referencing the old module name should use the new artifacts. This feature ensures both that we avoid multiple versions of the same library as well as doing conflict resolution across both projects.	
Enhanced Metadata Resolution Support	Dependency metadata can be modified after repository metadata is download but before it is chosen by Gradle as the final resolved version. This allows the creation of custom rules to do things like declare modules as changing (or snapshot) versions, or use a custom status scheme.	
Replacement of external and project dependencies	Dynamically replace external dependencies for project dependencies and vice versa. Especially helpful when only	

Feature	Gradle 3.4.1	Maven 3.3.3
	a subset of your modules are checked out locally.	
Custom Dependency Scopes	Don’t be limited by a predefined set of dependency scopes (compile, runtime, etc). Gradle allows you to define arbitrary dependency scopes. For example for integration tests that you may model in your build, to provision toolchains you need in your build, etc …	
Custom Repository Layout	Declare repositories with custom layouts. With custom layouts you can effectively treat nearly any file system directory structure as an artifact repository.	
File Based Dependencies	Not all dependencies are available from external repositories. Declare	

Feature	Gradle 3.4.1	Maven 3.3.3
	dependencies on filesystem resources when using a managed dependency isn't practical or when migrating legacy builds.	

Composite Build

Feature	Gradle 3.4.1	Maven 3.3.3
Combining separate builds	You can combine separate builds to work on code across multiple repositories.	
Ad-hoc composite build	You can use a command-line argument to include another build in the execution.	
Permanent composite build	You can define a composite build that permanently combines multiple builds.	
IDE import of a composite build		

Feature	Gradle 3.4.1	Maven 3.3.3
	Directly import a composite build into IntelliJ IDEA or Eclipse, or generate composite IDE projects with the built-in idea or eclipse tasks.	

Execution Model

Feature	Gradle 3.4.1	Maven 3.3.3
Fully Configurable DAG		For the Maven DAG a goal must depend on one and only one other goal if there is more than one goal associated with a lifecycle phase. Thus the goals of a lifecycle phase form an ordered list. The Maven DAG is only two levels deep. Those constraints prevent many domain scenarios from being properly modelled. Goals must depend on each other even though they have no dependencies. If a goal depends on more than one goal this must be mapped to an ordered list, destroying this relationship. This makes it often impossible to understand relationships from looking at the pom. This is true for humans as well as

Feature	Gradle 3.4.1	Maven 3.3.3
		for the build system itself which for example is not able to decide how to execute goals in parallel.
Task Exclusion	You can exclude any task from being run.	In Maven there is no generic exclude mechanism. Plugins have to implement it themselves.
Transitive Exclude	When you exclude a task, all tasks this task depends on are also automatically excluded if they have no other dependencies.	
Advanced Task Ordering	Beyond having full control about the dependencies that are created between tasks, Gradle has powerful language constructs to describe execution order between tasks even if tasks depends not on each others output. This can be modelled with shouldRunAfter and mustRunAfter relationships.	

Feature	Gradle 3.4.1	Maven 3.3.3
Finalizers	Tasks can be assigned to finalize another tasks similar to a finalizer clause in Java. They are always run after another task is executed, regardless whether this task fails or not. This is very powerful for example when doing lifecycle management for containers or databases.	
Task Dependency Inference	Gradle objects are aware of which tasks produce certain content. For example, the object representing the Java binary directory knows that the compile task produces the binaries. Any task that has the Java binary directory as input will automatically depend on the compile task. It does not need to be declared manually. This makes the build easier to maintain and more robust.	
True Multi Task Execution	You can specify multiple tasks when executing the	When you execute for example mvn clean compile

Feature	Gradle 3.4.1	Maven 3.3.3
	build and no task in the resulting DAG is executed twice. This is very helpful when you want to have an ad-hoc combination of behavior or when you deal with a build you can not change (for example, when configuring a CI job for the build). This is not possible with Ant or Maven.	test, the compile action is executed twice.
Continue Execution After Failures	Does not stop as soon as the first failure is encountered. Executes every task to be executed where all of the dependencies for that task completed without failure. Enables discovery of as many failures as possible in a single build execution with a very nice aggregated error report at the end.	The Maven reactor has a `--fail-at-end` option. It is less powerful compared to Gradle as it is not task based. It does not build a dependent module if anything went wrong in the module it depends on, even if the library of the module it depends on was properly produced.
Dry Run	Run a build to see which tasks actually get executed without executing the task actions.	

Feature	Gradle 3.4.1	Maven 3.3.3
Build Scans	A build scan is a shareable and centralized record of a build that provides insights into what happened and why.	
Build User Oriented Output Mode	The build output is very important part for the build user experience. In most other build tools the default output is geared towards a build author trying to debug a problem. This leads to a very verbose output hiding often important warnings and messages that are actually relevant for a developer running the build. Gradle's default output is geared towards a developer running the build and showing only messages that are relevant in this context and not abusing the log output as a poor man's progress indicator, for example when executing tests.	

Feature	Gradle 3.4.1	Maven 3.3.3
Rerouting output from external tools and libraries	The build output is very important for the build user experience. If you integrate with external tools and libraries their console output might be very verbose. In Gradle System.out and log output of Java Util Logging, Jakarta Commons Logging and Log4j is re-reouted to the Gradle logging system. You can define per external tool you are integrating with to which log level the output should be routed.	
Creating a custom logging schema	You can replace much of Gradle's logging UI with your own. You might do this, for example, if you want to customize the UI in some way, e.g. to log more or less information, or to change the formatting.	
Task Groups and Descriptions		

Feature	Gradle 3.4.1	Maven 3.3.3
	Every task can have a description and can be assigned to a group. This information is used for reporting and how the build is represented in the IDE.	Plugins and command can have descriptions. The grouping is the implicit grouping of the lifecycle phases.
Partial Transitive Multi-Module Builds	Build only the specified project and the ones it depends on.	
Automatic Validation of Task Inputs	Based on the input/output model Gradle automatically validates the configuration values of a task before it is executed.	

Configuration Model

Feature	Gradle 3.4.1	Maven 3.3.3
Declarative Builds	Declarative elements are elements that trigger the creation of executional elements. For example the Gradle Java plugin is a declarative element that when declared creates	Maven has one kind of declarative element, the packaging element. By specifying a certain type, e.g. jar, all the goals required to build a Java project are wired with the lifecycle phases.

Feature	Gradle 3.4.1	Maven 3.3.3
	Gradle tasks to model the full lifecycle of a Java project. Declarative elements are the 'What' of a Gradle build whereas the executional elements are the 'How'. Declarative elements often assume conventions when configuring the executional elements they create.	
Fine Grained Declarative Elements	Gradle has many fine grained declarative elements such as SourceSets or Android Product Flavors. They are at the heart of the Gradle DSL and make the Gradle build language so much richer then what you can do with Maven. They keep the build concise, easy to use, maintain and understand even if you have complex requirements. They provide a balance between structure and hiding unnecessary flexibility and enough flexibility so your developers can describe their requirements	Maven has no fine grained declarative elements. This is one major cause of the extreme inflexibility of Maven. Instead of providing you a rich, deeply integrated model you get a shallow, simplistic, overly rigid framework that is not able to deal with unanticipated requirements.
Custom Declarative Elements		

Feature	Gradle 3.4.1	Maven 3.3.3
	In Gradle, every plugin can contribute their own coarse or fine grained declarative elements. This enables you to provide a declarative approach even for your custom domains. It also allows every technology that integrates with Gradle to be modelled as a first class citizen and make it a pleasure to be used.	
Domain Objects Container	Every domain object describing your build, be it repositories, source directories, plugins or dependencies are stored in a responsive container that you can register listener with. You can write for example a company wide Gradle plugin that add rules to the repository container that no build script is allowed to add a repository that is not using a white list of URL's. You have full control over what particular builds scripts add to a build. You can react in any way. Augment or modify what has been added, let the build fail	

Feature	Gradle 3.4.1	Maven 3.3.3
	or issue a warning. You can add define dependencies that are only added for example if a build adds a particular plugin. A very, very powerful feature.	
Customizable Declarative Rules	Sometimes the default rules how declarative elements create executional elements don't tell the story you want. For example when declaring sources you might want a completely different toolchain than the one that is by default defined in Gradle and represented by the tasks Gradle creates as a result of the declaration. In the latest version of Gradle you can now change those rules. So far this is only available in the C/C++ and nightly Android support. It will we provide to Java later this year.	
Multiple Mix-Ins for Pre-Configuring Builds	In Gradle you can mix-in as many objects as you want to pre-configure your build via Gradle plugins. It is	In Maven you can define one and only object (Parent POM) that pre-configures your actual project. This is a

Feature	Gradle 3.4.1	Maven 3.3.3
	important to understand that Gradle plugins are conceptually very different from Maven plugins.	severe limitation to organize your build logic comprehensible and with DRY. Being able to mix-in multiple pom's into another pom is long-planned feature in the Maven world but it has not been released yet.
Fine Grained Build Event Listener	Gradle allows you to hook into every part of the build configuration and execution lifecycle for injecting custom behaviour, extracting information, adding additional logging and a tons of other use cases.	
User Based Behavior Injection	You can put custom listeners into your Gradle user home that hook into every Gradle build that is executed on your machine. With the lifecycle listeners described above you can add whatever custom behavior you want to individualize your build experience. For example adding and configuring the Gradle announcement plugin	

Feature	Gradle 3.4.1	Maven 3.3.3
	that pops up a window when the build is finishing or failing or adding a special repository that your are just using personally.	
Per Build Behavior Injection	Similar to user based behavior injection you can also specify on the command line additional listeners that hook into a build. This can be very helpful for example if you want your CI build to have specific behavior (e.g. fail if a non-standard repository is used).	
Custom Distributions	Every Gradle distribution has an init.d directory in which you can put custom scripts that pre-configure your build environment. You can use this to apply company wide custom rules that are enforced across all builds of all teams, to provide build-in set up tasks for developers, and so much more. Together with the Gradle wrapper you	

Feature	Gradle 3.4.1	Maven 3.3.3
	can easily distribute those custom distributions.	
Dynamic Task Creation	Sometimes you want to have a task whose behavior depends on a large or infinite number value range of parameters. A very nice and expressive way to provide such tasks are task rules.	
Custom Build Script Names	In Gradle you can change the name of the build scripts. For example in multi-module builds the build scripts of a subproject Foo can be named foo.gradle. That way you don’t loose orientation when you have many subproject build scripts open up in an editor.	

Build Infrastructure Administration

Feature	Gradle 3.4.1	Maven 3.3.3
Self Provisioning Build Environment		

Feature	Gradle 3.4.1	Maven 3.3.3
	With the Gradle wrapper the Gradle build environment is auto-provisioned. Furthermore you can determine which version should be used to build your project.	This is not supported by Maven core. There is an external Maven extension that has copied the Gradle Wrapper functionality. It has missing features like checksum support.
Version Controlled Build Environment Configuration	Important parameters for configuring the build environment can be stored in version as part of your project. No need for the developers to set them up manually. This includes the Gradle version to be used, the configuration for the JVM running the build and the JDK to be used for running the build.	
Custom Distributions	Every Gradle distribution has an init.d directory in which you can put custom scripts that pre-configure your build environment. You can use this to apply custom rules that are enforced, to provide	

Feature	Gradle 3.4.1	Maven 3.3.3
	build-in set up tasks for developers, and so much more. Together with the Gradle wrapper you can easily distribute those custom distributions.	

Continuous Build

Feature	Gradle 3.4.1	Maven 3.3.3
Build automatically when sources change	When a Gradle task is run in continuous mode, Gradle automatically watches for changes of the input of this task. Whenever the input changes, the task it automatically executed. You can run multiple tasks continuously in a multi-project build.	There's no global "watch" or "continuous" mode. Each plugin needs to add their own implementation if they want this functionality.
Trigger rules are automatically derived from the task definition		
Continuous build detects changes that occur during build execution		

Publishing Artifacts

Feature	Gradle 3.4.1	Maven 3.3.3
Generating and Publishing POM Metadata		
Generating and Publishing Ivy Metadata		
Publishing Multiple Artifacts	Gradle can publish multiple artifacts per project with different metadata. Be it an API and an implementation jar, a library and a test-fixture or variants for different Java platforms.	
Publish Custom Metadata	Gradle can not just use custom metadata for dependency resolution but can of course also publish this custom metadata, like status schemas, branches an artefact origins from, etc …	

Embedding

Feature	Gradle 3.4.1	Maven 3.3.3
SDK for embedded usage	Provided by the Gradle Tooling API	
Version agnostic		

Feature	Gradle 3.4.1	Maven 3.3.3
	The Gradle tooling API is back and forward compatible. With a particular version of the Tooling API you can drive build across all Gradle version since 1.0.	
Querying for Project model	You can query Gradle for the details of a build, including the project hierarchy and the project dependencies, external dependencies (including source and Javadoc jars), source directories and tasks of each project.	
Query for Build environment information	Gradle provides programmatically access to information about the build environment. This includes information about the Gradle Version, The Gradle User Home directory and the Java Home directory.	
Execute a build	You can execute a build and listen to stdout and stderr	

Feature	Gradle 3.4.1	Maven 3.3.3
	logging and progress (e.g. the stuff shown in the ‘status bar’ when you run on the command line).	
Build Operation Cancellation	All operations initiated via the Gradle Tooling API are gracefully cancellable at any time during the build	
Support custom JVM settings	JVM args provided via the Gradle Tooling API take precedence over gradle.properties	
Provide Eclipse Project Model	The Gradle Tooling API provides a model of how your project is mapped to an Eclipse project.	
Provide IDEA Project Model	The Gradle Tooling API provides a model how to your project is mapped to the IDEA model.	
Provide support for custom Project Model		

Feature	Gradle 3.4.1	Maven 3.3.3
	You can write a Gradle plugin to add custom metadata for the Gradle Tooling API. You can use this for example when you integrate your own product/customizations with Gradle.	
Run specific tests	The TestLauncher API allows running specific JUnit or TestNG Tests within one or more Test Tasks	
Register for progress events	Register for events of task or test progress to get informed about the process of the build with rich information about the processed task and test	
Run Continuous Builds	Gradle Tooling API provides the functionality to run a build programmatically in “continuous build” mode. Changes on build inputs (File changes) continuously trigger specified tasks to run.	

Ant Integration

Feature	Gradle 3.4.1	Maven 3.3.3
Run Targets from Ant Build		
Embed Ant Tasks	You can deeply integrate any default, optional or cutom Ant task.	
Deep Import of Ant Builds	In Gradle you can import an Ant Build at runtime. All the Ant targets are wrapped around Gradle tasks and thus fully integrated. You can for example enable incremental and parallel builds for them and other Gradle tasks can depend on them. It is also possible to read configuration information from your Ant build or inject configuration from Gradle into your Ant build.	
Partial Replacement of Ant Targets with Gradle Tasks	You can even make your Ant targets dependent on Gradle tasks. This allows you to fully replace some part of your Ant build towards Gradle and import the other part.	

Android Support

Feature	Gradle 3.4.1	Maven 3.3.3
Support for Building Android Applications		
Maintained by the Google Android Team	The Google Android team is the owner and maintainer of the Gradle Android Plugin.	
Support for official Android Project Layout and Conventions		
Android Domain Build Language	Describe your project via an Android-specific DSL. Tell Gradle about your project, now how to build it. Avoid the overly verbose need to explicitly implement the complex build process of Android projects.	
Full Integration with Android Studio	Gradle is deeply integrated with Android Studio, the official Android IDE. In fact, Android Studio has no internal builder, it instead delegates all build tasks to Gradle. This “unified build system” ensures correctness across all your builds,	

Feature	Gradle 3.4.1	Maven 3.3.3
	whether they be run from Android Studio, the command line, or a continuous integration build server.	
Build Variants	Declare your build types and product flavors and let Gradle handle the rest. No need to copy and paste build logic over and over for each of your project's variants. Use the expressive DSL to declare only what is different about your variants.	
Android Library Projects	Android library projects are treated as first-class citizens in Gradle's multi-project build model. This enables all the benefits provided by Gradle's multi-project build support, such as project dependencies, and incremental builds.	No build type of product flavor support.
Manifest Merging	Manifests and resource files are automatically merged, and individual entries overridden from variant	Since the Maven Android plugin lacks support for modeling build variants, manifest and resource

Feature	Gradle 3.4.1	Maven 3.3.3
	source files. Build APKs with different application names, SKD versions or even required permissions.	merging must be configured explicitly.
Per-Variant Dependency Management	Gradle's advanced dependency management features are available to Android projects and can be configured uniquely for each build variant. Declare dependencies relevant to only certain variants or customize dependency resolution for each of your build variants.	
Application Signing	Automate signing your applications. Debug build variants are, by default, signed with a debug key for installation on development devices. Declare additional signing configurations for publication to the Google Play store.	
ProGuard Support	Easily configure ProGuard on your project to enable obfuscation and minification	

Feature	Gradle 3.4.1	Maven 3.3.3
	of the built APK. Configure ProGuard independently for each build type, allow an unoptimized APK for development and optimized one for release.	
Unit Testing	Run unit tests against your Android application or library. Unit tests run in a standard Java JVM against a mocked Android SDK implementation facilitating fast-feedback test development, eliminating the overhead of building a separate APK and running tests of a device or emulator.	
On-device Functional Testing	For tests that require a real Android environment for proper execution, bundle your test in a separate APK to be installed and run on an Android device or emulator. Separate APKs are built for each build variant allowing you to build and test every variant of your app in a single Gradle build.	Test code must live in a separate project causing unnecessary workspace bloat.
APK Splits		

Feature	Gradle 3.4.1	Maven 3.3.3
	Efficiently build multiple APKs for differing device display densities or ABIs by configuring Gradle to reuse shared build outputs.	
Multidex Support	Utilize multidex support to avoid the 65k method limit imposed by Android DEX files.	
NDK Support	Integrate your build with the Android NDK. Build application that depend on native libraries, projects that compile C/C++ into native libraries or both.	
Databinding	Using binding classes generated by your Gradle build you can drastically simplify the way you interact with Android views. No more interacting with views directly, simply update the backing object and your view is kept up to date.	

Native Language Support

Feature	Gradle 3.4.1	Maven 3.3.3
Build C/C++/Obj-C/Obj-C++/Assembler	Gradle has built in support for compiling and linking programs using Assembler, C/C++ and Obj-C/C++. Gradle can build shared and static libraries and executables.	There is native support for Maven via the Maven NAR plugin. We have not evaluated it in depth and therefore can't compare it yet with Gradle's native support. A contribution that does this comparison would be much appreciated.
Model variants of a native component	Easily model variants of a native component like support for different ABI's, OS, build types, etc.	
Supports GCC	Gradle supports building with GCC4 on Linux, Windows (with Cygwin and MingW) and Mac OS X.	
Supports Clang	Gradle supports building with Clang on Linux and Mac OS X.	

Feature	Gradle 3.4.1	Maven 3.3.3
Supports MS Visual C++	Gradle supports building with Microsoft's Visual C++ compiler on Windows. (VS 2010 and VS 2013 supported)	
Generates Windows Resources	Gradle uses Microsoft's resource compiler to build Windows resource script files into your application.	
Parallel Compilation	When building native code, Gradle divides the compilation step into parallelizable tasks and executes them in a shared resource pool. This speeds up the single project case and ensures that Gradle does not consume too many resources in parallel multi-project builds.	
Precompiled Headers		

Feature	Gradle 3.4.1	Maven 3.3.3
	Gradle makes it easy to use precompiled headers when building your software. Precompiled headers can speed up compilation times if your project has many header files that are included in most of your source code. Precompiled headers is a compiler-specific optimization to cache an already parsed version of your headers.	
Build Shared/Static Libraries, Executables		
Build mixed language binaries	Gradle can build separate languages (e.g., Assembler and C) and link them into a single executable or library.	
CUnit Test Support	Gradle supports testing C applications with CUnit.	
GoogleTest Support	Gradle supports testing C++ applications with GoogleTest.	

Feature	Gradle 3.4.1	Maven 3.3.3
Makefile support	Gradle does not come with built-in support for extracting a build from a Makefile, but Gradle's Exec task can be used to wrap and existing Makefile when migrating to Gradle.	
Build JNI Libraries	Gradle does not come with an out-of-the-box recipe for building a JNI library, but you can use a custom task to generate the headers and build a Shared Library as usual.	
Dependency Management	You can use the current Gradle dependency management support to support binary sharing. But it is not fully tailored yet for the needs of native domain. Soon our new variant aware dependency management will provide the first full solution for dependency management in the native world.	

Functional Testing and Build Logic

Feature	Gradle 3.4.1	Maven 3.3.3
Programmatic execution of build through API agnostic of test framework	Builds are executed using the Tooling API.	
Inspection of build outcome and output (e.g. result of build, standard out/error, executed tasks etc.)		
Cross-version compatibility testing	Usage of any Gradle distribution installed locally or available on server.	
Debugging the build under test from the IDE	Any IDE that supports injection of JVM system properties for test execution.	

Other Features

Feature	Gradle 3.4.1	Maven 3.3.3
Task Name Abbreviation when executing	You can call a build action by abbreviations. For example if you type gradle uA it will call	

Feature	Gradle 3.4.1	Maven 3.3.3
	the task uploadArchives if there are not naming ambiguities.	
Fast Release Cycles And High Activity	Since the release of Gradle 1.0 in June 2012, there have been 30 minor or major releases. Gradle releases a new version usually every 6-8 weeks. Most minor releases in Gradle add a significant set of new features.	Since the release of Maven 2 in October 2005 there have been only 5 major or minor releases. The minor releases have been mostly bug fix releases with hardly any new functionality.