Upgrading Packages

Sui smart contracts are immutable package objects consisting of a collection of Move modules. Because the packages are immutable, transactions can safely access smart contracts without full consensus (fastpath transactions). If someone could change these packages, they would become shared objects , which would require full consensus before completing a transaction.

The inability to change package objects, however, becomes a problem when considering the iterative nature of code development. Builders require the ability to update their code and pull changes from other developers while still being able to reap the benefits of fastpath transactions. Fortunately, the Sui network provides a method of upgrading your packages while still retaining their immutable properties.

There are some details of the process that you should consider before upgrading your packages.

For example, module initializers do not re-run with package upgrades. When you publish your initial package, Move runs the init function you define for the package once (and only once) at the time of the publish event. Any init functions you might include in subsequent versions of your package are ignored. See Module Initializer in The Move Book for more information.

As alluded to previously, all packages on the Sui network are immutable. Because of this fact, you cannot delete old packages from the chain. As a result, there is nothing that prevents other packages from accessing the methods and types defined in the old versions of your upgraded packages. By default, users can choose to keep using the old version of a package, as well. As a package developer, you must be aware of and account for this possibility.

For example, you might define an increment function in your original package:

Then, your package upgrade might add an emit event to the increment function:

If there is a mix of callers for both the old and upgraded increment function, then the process fails because the old function is not aware of the Progress event.

Similar to mismatched function definitions, you might also run into issues maintaining dynamic fields that need to remain in sync with a struct's original fields. To address these issues, you can introduce a new type as part of the upgrade and port users over to it, breaking backwards compatibility. For example, if you're using owned objects to demonstrate proof, like proof of ownership, and you develop a new version of your package to address problematic code, you can introduce a new type in the upgraded package. You can then add a function to your package that trades old objects for new ones. Because your logic only recognizes objects with the new type, you effectively force users to update.

Another example of having users update to the latest package is when you have a bookkeeping shared object in your package that you discover has flawed logic so is not functioning as expected. As in the previous example, you want users to use only the object defined in the upgraded package with the correct logic, so you add a new type and migration function to your package upgrade. This process requires a couple of transactions, one for the upgrade and another that you call from the upgraded package to set up the new shared object that replaces the existing one. To protect the setup function, you would need to create an AdminCap object or similar as part of your package to make sure you, as the package owner, are the only authorized initiator of that function. Perhaps even more useful, you might include a flag in the shared object that allows you, as the package owner, to toggle the enabled state of that shared object. You can add a check for the enabled state to prevent access to that object from the on-chain public while you perform the migration. Of course, you would probably create this flag only if you expected to perform this migration at some point in the future, not because you're intentionally developing objects with flawed logic.

When you create packages that involve shared objects, you need to think about upgrades and versioning from the start given that all prior versions of a package still exist on-chain . A useful pattern is to introduce versioning to the shared object and using a version check to guard access to functions in the package. This enables you to limit access to the shared object to only the latest version of a package.

Considering the earlier counter example, which might have started life as follows:

To ensure that upgrades to this package can limit access of the shared object to the latest version of the package, you need to:

An upgrade-aware counter module that incorporates all these ideas looks as follows:

To upgrade a module using this pattern requires making two extra changes, on top of any implementation changes your upgrade requires:

The following module is an upgraded counter that emits Progress events as originally discussed, but also provides tools for an admin (AdminCap holder) to prevent accesses to the counter from older package versions:

Upgrading to this version of the package requires performing the package upgrade, and calling the migrate function in a follow-up transaction. Note that the migrate function:

After a successful upgrade, calls to increment on the previous version of the package aborts on the version check, while calls on the later version should succeed.

This pattern forms the basis for upgradeable packages involving shared objects, but you can extend it in a number of ways, depending on your package's needs:

To upgrade a package, your package must satisfy the following requirements:

If you have a package with a dependency, and that dependency is upgraded, your package does not automatically depend on the newer version. You must explicitly upgrade your own package to point to the new dependency.

Use the sui client upgrade command to upgrade packages that meet the previous requirements, providing values for the following flags:

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Developers upgrading packages using Move code have access to types and functions to define custom upgrade policies. For example, a Move developer might want to disallow upgrading a package, regardless of the current package owner. The make_immutable function is available to them to create this behavior. More advanced policies using available types like UpgradeTicket and Upgrade Receipt are also possible. For an example, see this custom upgrade policy on GitHub.

When you use the Sui Client CLI, the upgrade command handles generating the upgrade digest, authorizing the upgrade with the UpgradeCap to get an UpgradeTicket , and updating the UpgradeCap with the UpgradeReceipt after a successful upgrade. To learn more about these processes, see the Move documentation for the package module .

You develop a package named sui_package . Its manifest looks like the following:

When your package is ready, you publish it:

And receive the response:

The result includes an Object changes section with two pieces of information you need for upgrading, an UpgradeCap ID and your package ID.

You can identify the different objects using the Object.objectType value in the response. The UpgradeCap entry has a value of String("0x2::package::UpgradeCap") and the objectType for the package reads String("::sui_package::")

Beginning with the Sui v1.29.0 release, published addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you can follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

After a while, you decide to upgrade your sui_package to include some requested features.

If your package has not adopted automated address management you'll need to take the following manual steps.

To make sure your other packages can use this package as a dependency, you must update the Move.toml manifest file for your package to include published information.

Update the alias address and add a new published-at entry in the [package] section, both pointing to the value of the on-chain ID:

After a while, you decide to upgrade your sui_package to include some requested features. Before running the upgrade command, you need to edit the manifest again.

In the [addresses] section, you update the sui_package address value to 0x0 again so the validator issues a new address for the upgrade package. You can leave the published-at value the same, because it is only read by the toolchain when publishing a dependent package. The saved manifest now resembles the following:

With the new manifest and code in place, you can proceed.

Run sui client upgrade command to upgrade your package. Pass the UpgradeCap ID (the value from the example) to the --upgrade-

capability flag.

The console alerts you if the new package doesn't satisfy requirements , otherwise the compiler publishes the upgraded package to the network and returns its result:

The result provides a new ID for the upgraded package.

Beginning with the Sui v1.29.0 release, upgraded addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you may follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

So that packages that depend on your sui_package know where to find the on-chain bytecode for verification, edit your manifest again. Provide the upgraded package ID for the published-at value and return the original sui_package ID value in the [addresses] section:

Note the published-at value changes with every upgrade and needs to be updated after every upgrade.

The ID for the sui_package in the [addresses] section always points to the original package ID after upgrading. You must always change that value back to 0x0 , however, before running the upgrade command so the validator knows to create a new ID for the upgrade.

Upgrade considerations

There are some details of the process that you should consider before upgrading your packages.

For example, module initializers do not re-run with package upgrades. When you publish your initial package, Move runs the init function you define for the package once (and only once) at the time of the publish event. Any init functions you might include in subsequent versions of your package are ignored. See Module Initializer in The Move Book for more information.

As alluded to previously, all packages on the Sui network are immutable. Because of this fact, you cannot delete old packages from the chain. As a result, there is nothing that prevents other packages from accessing the methods and types defined in the old versions of your upgraded packages. By default, users can choose to keep using the old version of a package, as well. As a package developer, you must be aware of and account for this possibility.

For example, you might define an increment function in your original package:

Then, your package upgrade might add an emit event to the increment function:

If there is a mix of callers for both the old and upgraded increment function, then the process fails because the old function is not aware of the Progress event.

Similar to mismatched function definitions, you might also run into issues maintaining dynamic fields that need to remain in sync with a struct's original fields. To address these issues, you can introduce a new type as part of the upgrade and port users over to it, breaking backwards compatibility. For example, if you're using owned objects to demonstrate proof, like proof of ownership, and you develop a new version of your package to address problematic code, you can introduce a new type in the upgraded package. You can then add a function to your package that trades old objects for new ones. Because your logic only recognizes objects with the new type, you effectively force users to update.

Another example of having users update to the latest package is when you have a bookkeeping shared object in your package that you discover has flawed logic so is not functioning as expected. As in the previous example, you want users to use only the object defined in the upgraded package with the correct logic, so you add a new type and migration function to your package upgrade. This process requires a couple of transactions, one for the upgrade and another that you call from the upgraded package to set up the new shared object that replaces the existing one. To protect the setup function, you would need to create an AdminCap object or similar as part of your package to make sure you, as the package owner, are the only authorized initiator of that function. Perhaps even more useful, you might include a flag in the shared object that allows you, as the package owner, to toggle the enabled state of that shared object. You can add a check for the enabled state to prevent access to that object from the on-chain public while you perform the migration. Of course, you would probably create this flag only if you expected to perform this migration at some point in the future, not because you're intentionally developing objects with flawed logic.

When you create packages that involve shared objects, you need to think about upgrades and versioning from the start given that all prior versions of a package still exist on-chain . A useful pattern is to introduce versioning to the shared object and using a version

check to guard access to functions in the package. This enables you to limit access to the shared object to only the latest version of a package.

Considering the earlier counter example, which might have started life as follows:

To ensure that upgrades to this package can limit access of the shared object to the latest version of the package, you need to:

An upgrade-aware counter module that incorporates all these ideas looks as follows:

To upgrade a module using this pattern requires making two extra changes, on top of any implementation changes your upgrade requires:

The following module is an upgraded counter that emits Progress events as originally discussed, but also provides tools for an admin (AdminCap holder) to prevent accesses to the counter from older package versions:

Upgrading to this version of the package requires performing the package upgrade, and calling the migrate function in a follow-up transaction. Note that the migrate function:

After a successful upgrade, calls to increment on the previous version of the package aborts on the version check, while calls on the later version should succeed.

This pattern forms the basis for upgradeable packages involving shared objects, but you can extend it in a number of ways, depending on your package's needs:

To upgrade a package, your package must satisfy the following requirements:

If you have a package with a dependency, and that dependency is upgraded, your package does not automatically depend on the newer version. You must explicitly upgrade your own package to point to the new dependency.

Use the sui client upgrade command to upgrade packages that meet the previous requirements, providing values for the following flags:

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Developers upgrading packages using Move code have access to types and functions to define custom upgrade policies. For example, a Move developer might want to disallow upgrading a package, regardless of the current package owner. The make_immutable function is available to them to create this behavior. More advanced policies using available types like UpgradeTicket and Upgrade Receipt are also possible. For an example, see this custom upgrade policy on GitHub.

When you use the Sui Client CLI, the upgrade command handles generating the upgrade digest, authorizing the upgrade with the UpgradeCap to get an UpgradeTicket , and updating the UpgradeCap with the UpgradeReceipt after a successful upgrade. To learn more about these processes, see the Move documentation for the package module .

You develop a package named sui_package . Its manifest looks like the following:

When your package is ready, you publish it:

And receive the response:

The result includes an Object changes section with two pieces of information you need for upgrading, an UpgradeCap ID and your package ID.

You can identify the different objects using the Object.objectType value in the response. The UpgradeCap entry has a value of String("0x2::package::UpgradeCap") and the objectType for the package reads String("::sui_package::")

Beginning with the Sui v1.29.0 release, published addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you can follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

After a while, you decide to upgrade your sui_package to include some requested features.

If your package has not adopted automated address management you'll need to take the following manual steps.

To make sure your other packages can use this package as a dependency, you must update the Move.toml manifest file for your

package to include published information.

Update the alias address and add a new published-at entry in the [package] section, both pointing to the value of the on-chain ID:

After a while, you decide to upgrade your sui_package to include some requested features. Before running the upgrade command, you need to edit the manifest again.

In the [addresses] section, you update the sui_package address value to 0x0 again so the validator issues a new address for the upgrade package. You can leave the published-at value the same, because it is only read by the toolchain when publishing a dependent package. The saved manifest now resembles the following:

With the new manifest and code in place, you can proceed.

Run sui client upgrade command to upgrade your package. Pass the UpgradeCap ID (the value from the example) to the --upgrade-capability flag.

The console alerts you if the new package doesn't satisfy requirements , otherwise the compiler publishes the upgraded package to the network and returns its result:

The result provides a new ID for the upgraded package.

Beginning with the Sui v1.29.0 release, upgraded addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you may follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

So that packages that depend on your sui_package know where to find the on-chain bytecode for verification, edit your manifest again. Provide the upgraded package ID for the published-at value and return the original sui_package ID value in the [addresses] section:

Note the published-at value changes with every upgrade and needs to be updated after every upgrade.

The ID for the sui_package in the [addresses] section always points to the original package ID after upgrading. You must always change that value back to 0x0 , however, before running the upgrade command so the validator knows to create a new ID for the upgrade.

Upgrade requirements

To upgrade a package, your package must satisfy the following requirements:

If you have a package with a dependency, and that dependency is upgraded, your package does not automatically depend on the newer version. You must explicitly upgrade your own package to point to the new dependency.

Use the sui client upgrade command to upgrade packages that meet the previous requirements, providing values for the following flags:

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Developers upgrading packages using Move code have access to types and functions to define custom upgrade policies. For example, a Move developer might want to disallow upgrading a package, regardless of the current package owner. The make_immutable function is available to them to create this behavior. More advanced policies using available types like UpgradeTicket and Upgrade Receipt are also possible. For an example, see this custom upgrade policy on GitHub.

When you use the Sui Client CLI, the upgrade command handles generating the upgrade digest, authorizing the upgrade with the UpgradeCap to get an UpgradeTicket , and updating the UpgradeCap with the UpgradeReceipt after a successful upgrade. To learn more about these processes, see the Move documentation for the package module .

You develop a package named sui_package . Its manifest looks like the following:

When your package is ready, you publish it:

And receive the response:

The result includes an Object changes section with two pieces of information you need for upgrading, an UpgradeCap ID and your

package ID.

You can identify the different objects using the Object.objectType value in the response. The UpgradeCap entry has a value of String("0x2::package::UpgradeCap") and the objectType for the package reads String("::sui_package::")

Beginning with the Sui v1.29.0 release, published addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you can follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

After a while, you decide to upgrade your sui_package to include some requested features.

If your package has not adopted automated address management you'll need to take the following manual steps.

To make sure your other packages can use this package as a dependency, you must update the Move.toml manifest file for your package to include published information.

Update the alias address and add a new published-at entry in the [package] section, both pointing to the value of the on-chain ID:

After a while, you decide to upgrade your sui_package to include some requested features. Before running the upgrade command, you need to edit the manifest again.

In the [addresses] section, you update the sui_package address value to 0x0 again so the validator issues a new address for the upgrade package. You can leave the published-at value the same, because it is only read by the toolchain when publishing a dependent package. The saved manifest now resembles the following:

With the new manifest and code in place, you can proceed.

Run sui client upgrade command to upgrade your package. Pass the UpgradeCap ID (the value from the example) to the --upgrade-capability flag.

The console alerts you if the new package doesn't satisfy requirements , otherwise the compiler publishes the upgraded package to the network and returns its result:

The result provides a new ID for the upgraded package.

Beginning with the Sui v1.29.0 release, upgraded addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you may follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

So that packages that depend on your sui_package know where to find the on-chain bytecode for verification, edit your manifest again. Provide the upgraded package ID for the published-at value and return the original sui_package ID value in the [addresses] section:

Note the published-at value changes with every upgrade and needs to be updated after every upgrade.

The ID for the sui_package in the [addresses] section always points to the original package ID after upgrading. You must always change that value back to 0x0 , however, before running the upgrade command so the validator knows to create a new ID for the upgrade.

Upgrading

Use the sui client upgrade command to upgrade packages that meet the previous requirements, providing values for the following flags:

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

Developers upgrading packages using Move code have access to types and functions to define custom upgrade policies. For example, a Move developer might want to disallow upgrading a package, regardless of the current package owner. The make_immutable function is available to them to create this behavior. More advanced policies using available types like UpgradeTicket and Upgrade Receipt are also possible. For an example, see this custom upgrade policy on GitHub.

When you use the Sui Client CLI, the upgrade command handles generating the upgrade digest, authorizing the upgrade with the UpgradeCap to get an UpgradeTicket , and updating the UpgradeCap with the UpgradeReceipt after a successful upgrade. To learn more about these processes, see the Move documentation for the package module .

You develop a package named sui_package . Its manifest looks like the following:

When your package is ready, you publish it:

And receive the response:

The result includes an Object changes section with two pieces of information you need for upgrading, an UpgradeCap ID and your package ID.

You can identify the different objects using the Object.objectType value in the response. The UpgradeCap entry has a value of String("0x2::package::UpgradeCap") and the objectType for the package reads String("::sui_package::")

Beginning with the Sui v1.29.0 release, published addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you can follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

After a while, you decide to upgrade your sui_package to include some requested features.

If your package has not adopted automated address management you'll need to take the following manual steps.

To make sure your other packages can use this package as a dependency, you must update the Move.toml manifest file for your package to include published information.

Update the alias address and add a new published-at entry in the [package] section, both pointing to the value of the on-chain ID:

After a while, you decide to upgrade your sui_package to include some requested features. Before running the upgrade command, you need to edit the manifest again.

In the [addresses] section, you update the sui_package address value to 0x0 again so the validator issues a new address for the upgrade package. You can leave the published-at value the same, because it is only read by the toolchain when publishing a dependent package. The saved manifest now resembles the following:

With the new manifest and code in place, you can proceed.

Run sui client upgrade command to upgrade your package. Pass the UpgradeCap ID (the value from the example) to the --upgrade-capability flag.

The console alerts you if the new package doesn't satisfy requirements , otherwise the compiler publishes the upgraded package to the network and returns its result:

The result provides a new ID for the upgraded package.

Beginning with the Sui v1.29.0 release, upgraded addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you may follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

So that packages that depend on your sui_package know where to find the on-chain bytecode for verification, edit your manifest again. Provide the upgraded package ID for the published-at value and return the original sui_package ID value in the [addresses] section:

Note the published-at value changes with every upgrade and needs to be updated after every upgrade.

The ID for the sui_package in the [addresses] section always points to the original package ID after upgrading. You must always change that value back to 0x0 , however, before running the upgrade command so the validator knows to create a new ID for the upgrade.

Example

You develop a package named sui_package . Its manifest looks like the following:

When your package is ready, you publish it:

And receive the response:

The result includes an Object changes section with two pieces of information you need for upgrading, an UpgradeCap ID and your package ID.

You can identify the different objects using the Object.objectType value in the response. The UpgradeCap entry has a value of String("0x2::package::UpgradeCap") and the objectType for the package reads String("::sui_package::")

Beginning with the Sui v1.29.0 release, published addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you can follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

After a while, you decide to upgrade your sui_package to include some requested features.

If your package has not adopted automated address management you'll need to take the following manual steps.

To make sure your other packages can use this package as a dependency, you must update the Move.toml manifest file for your package to include published information.

Update the alias address and add a new published-at entry in the [package] section, both pointing to the value of the on-chain ID:

After a while, you decide to upgrade your sui_package to include some requested features. Before running the upgrade command, you need to edit the manifest again.

In the [addresses] section, you update the sui_package address value to 0x0 again so the validator issues a new address for the upgrade package. You can leave the published-at value the same, because it is only read by the toolchain when publishing a dependent package. The saved manifest now resembles the following:

With the new manifest and code in place, you can proceed.

Run sui client upgrade command to upgrade your package. Pass the UpgradeCap ID (the value from the example) to the --upgrade-capability flag.

The console alerts you if the new package doesn't satisfy requirements , otherwise the compiler publishes the upgraded package to the network and returns its result:

The result provides a new ID for the upgraded package.

Beginning with the Sui v1.29.0 release, upgraded addresses are automatically managed in the Move.lock file and you do not need to take further action.

If the package was published or upgraded with a Sui version prior to v1.29.0 , you may follow the guide for adopting automated address management . Alternatively, refer to the Manual Addresses tab above for further steps.

So that packages that depend on your sui_package know where to find the on-chain bytecode for verification, edit your manifest again. Provide the upgraded package ID for the published-at value and return the original sui_package ID value in the [addresses] section:

Note the published-at value changes with every upgrade and needs to be updated after every upgrade.

The ID for the sui_package in the [addresses] section always points to the original package ID after upgrading. You must always change that value back to 0x0 , however, before running the upgrade command so the validator knows to create a new ID for the upgrade.