Deep Dive into Declarative OS Updates & Upgrades

Catherine Davis @ MacSysAdmin Sweden 2025

Agenda

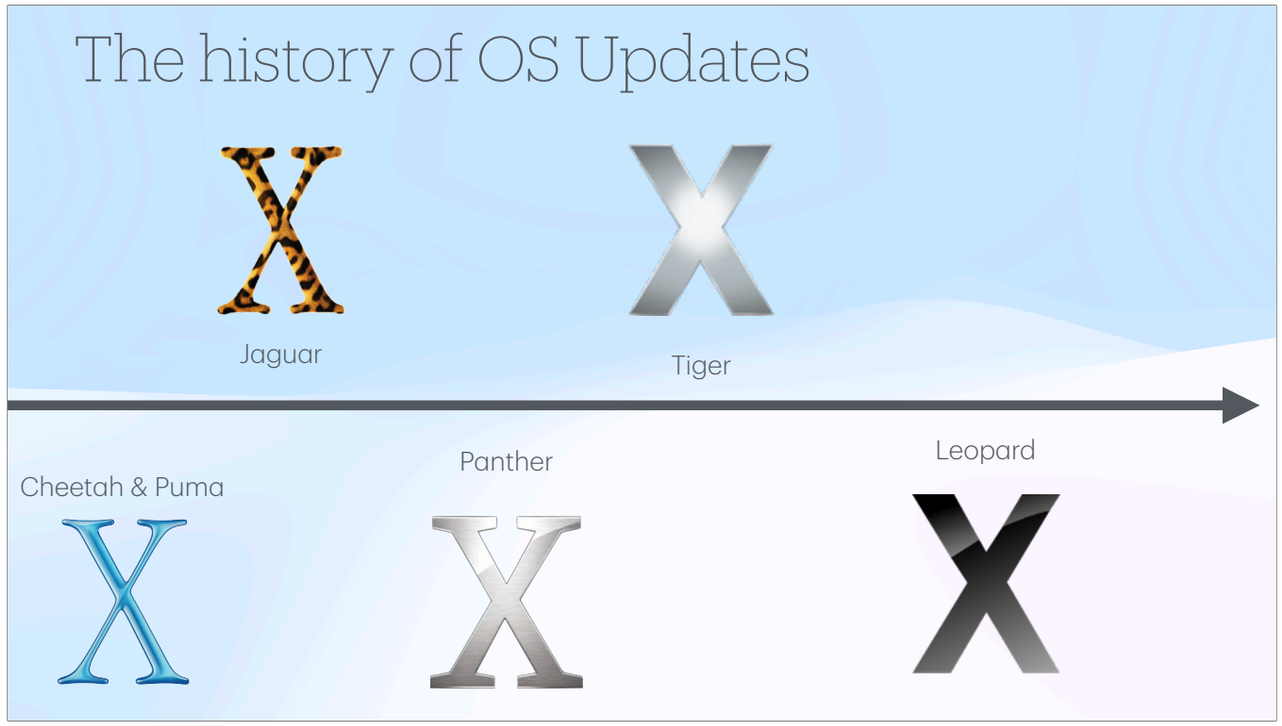
A Brief History

The Updating Experience

Debugging

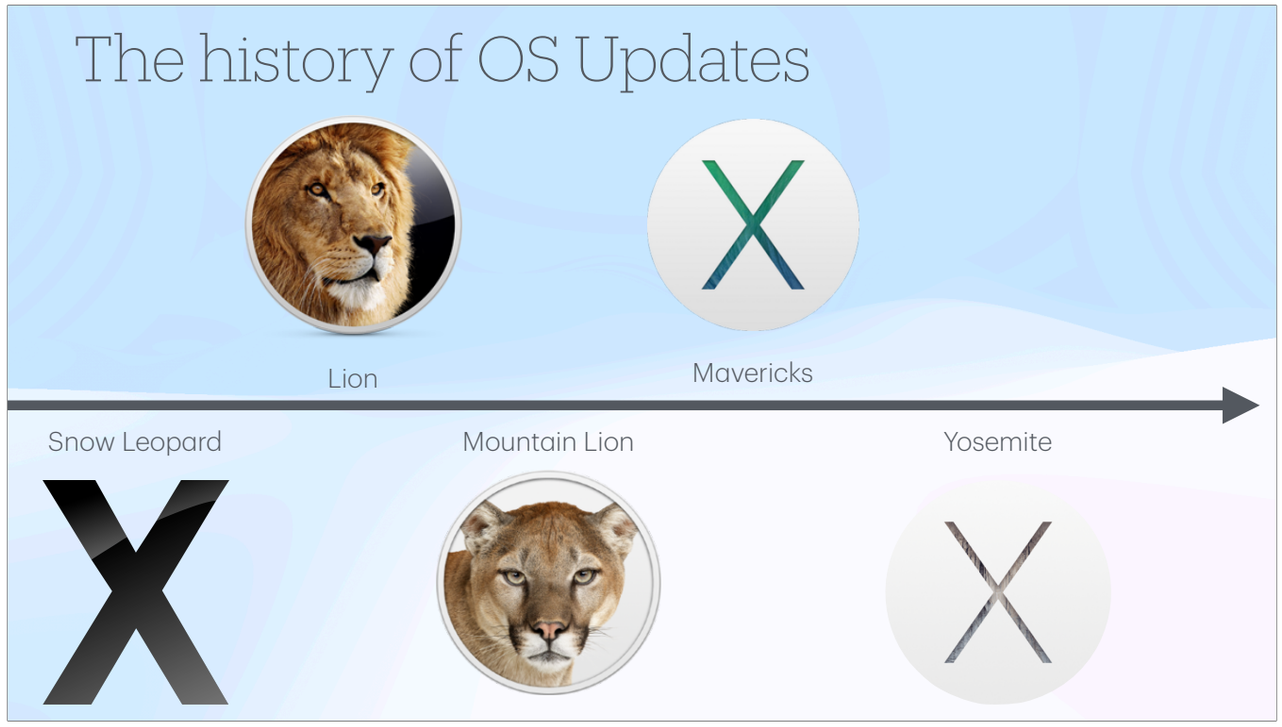
Feedback

For today’s session, I’ll be going through a brief history of updates through the ages, because understanding the evolution is important to know how and why we got to where we are today. I’ll also be covering the update experience, and then diving in to a bit of debugging information before capping it all off with some tips on how you can provide feedback to both Apple and your MDM provider.



Stepping way back in time now, all the way to my family’s first Mac running Cheetah, macOS 10.0 - now personally, I was in middle school and didn’t care about OS updates, but NOW I do, and the history is worth knowing.

There were a couple different ways you could do the updates for these early versions, whether through System Preferences or using the Software Update command line, or you could use a fun thing known as Combo Update packages, and what those would do is if you wanted to jump from like 10.4.2 or whatever you had on your DVD to say 10.4.11 as your final version, you could use the Combo Updater to do everything over the top of that for you.

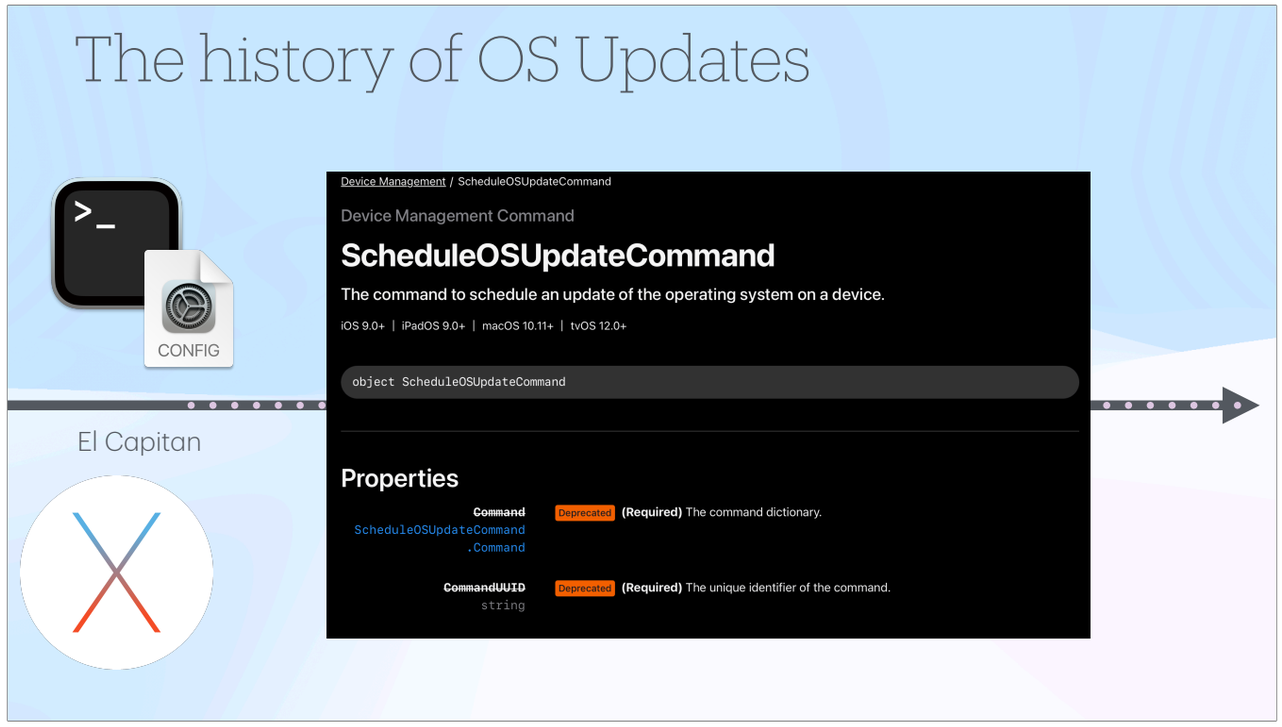


Then we jump ahead to Snow Leopard, which was the last version that was shipped on a disc.

Then we have Lion, and this was the version that was out when I started as an Admin, but more importantly, it’s the first version where the only way to get was from the App Store with the full installer. You could still use the System Update pane to update, there’s still the combo updaters, there’s still the software update binary. Now though with the App Store you could get the full versions for each of the major and minor versions and you could use that start OS Install binary within it and do the updates that way.

Now we jump forward to 2015 with El Capitan, and with it we have our first MDM updates using the schedule OS Update, available OS Update, and the OS Update Status commands.

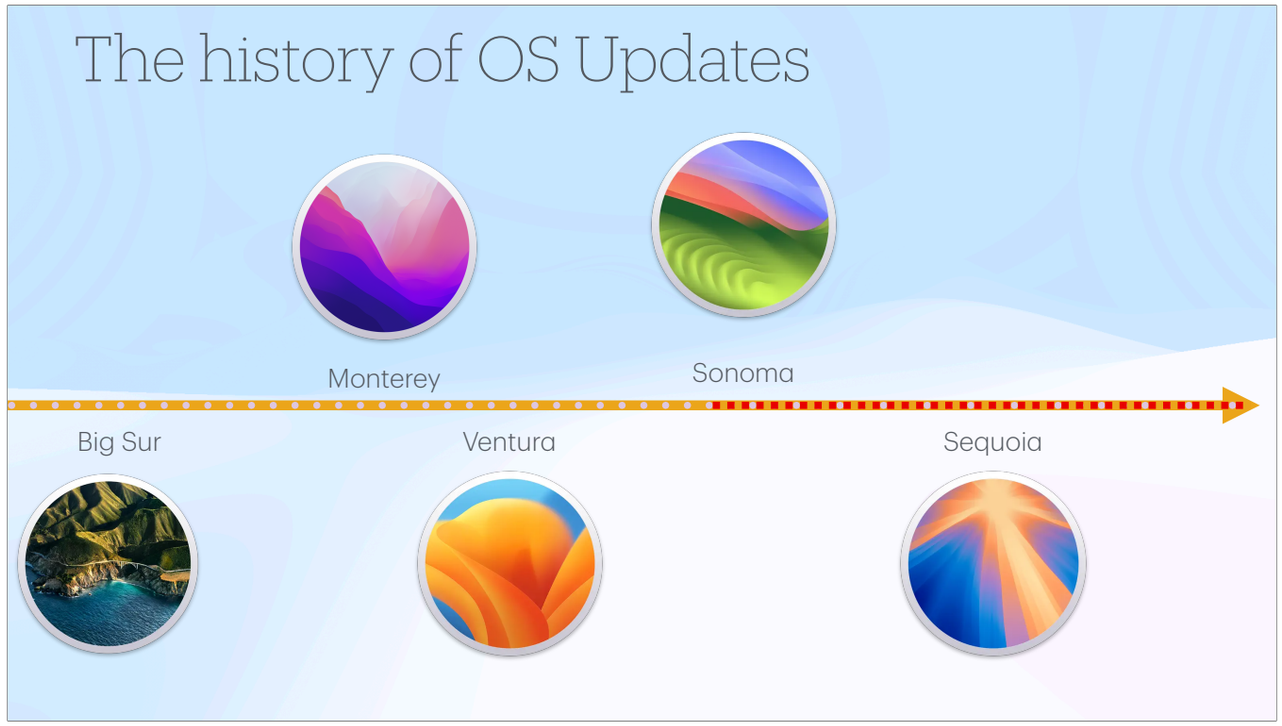
At this point you could still use System Preferences pane, or the App Store, or the Software Update command line installer, and you can still use the combo update packages, so basically we're just adding more and more ways to do OS Updates, but haven't really started deprecating methods YET.



However - this method of OS Update is now considered deprecated as of WWDC this year. The note in the public documentation is that it is deprecated and you should start using Declarative in the near future.

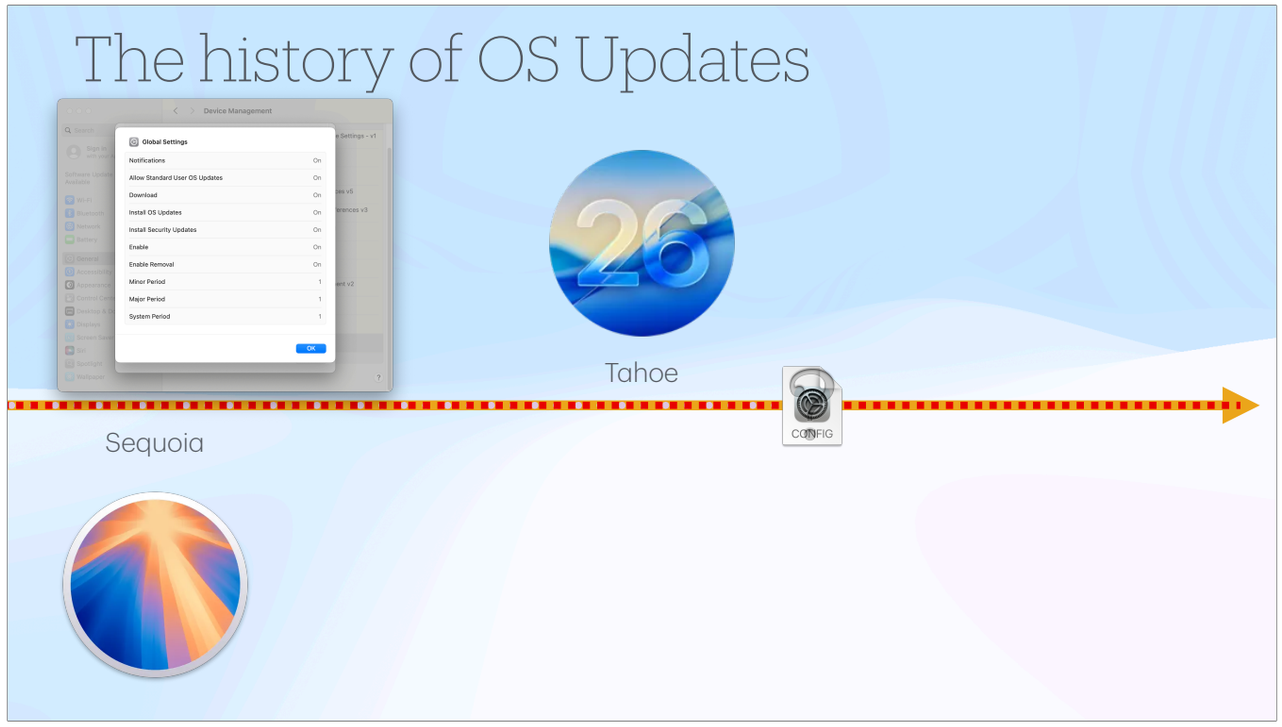
Now we go back to the timeline and get to Catalina, and this is the end of the road for those combo updaters. There is a good reason for that though, with 10.11, the Big Sur release, those installers function differently. The updates now only changed what was needed, and eliminated some of those extra read/write cycles as we saw the SSDs getting rolled out on Macs.

Big Sur also caused some confusion in the updates world, because it’s also the first version that requires local authentication for updates or upgrades from the volume owner, or an authenticated reboot using the bootstrap via an MDM command.



Now we’ll keep heading along until we hit the next big shift, which is Sonoma.

This is where we start to add in the enforcement specific declaration. It's also when Apple added the 403 response within MDM and ADE enrollment where you could have a device update before enrolling in the MDM. Then with macOS 15, we got the ProPlus MAX trademark, which is like another iteration on top of enforcement specific declaration, it gave us a new declaration object with more granular controls around your update cadence settings. This is really modernizing the config profile where you had deferrals, this is the declaration version of that with some additional controls.



Now, 14 slides in, we're at current era. I mentioned a little bit ago that in WWDC 2025, Apple states that software update management using mobile device management commands restrictions and the aforementioned profile payload and its queries is deprecated. Apple will remove it next year. Going forward organizations can manage and enforce software updates using only declarative management. At the end, I'll discuss how the community can make sure Apple is getting the right feedback and make sure all use cases are being covered before that switch is flipped.

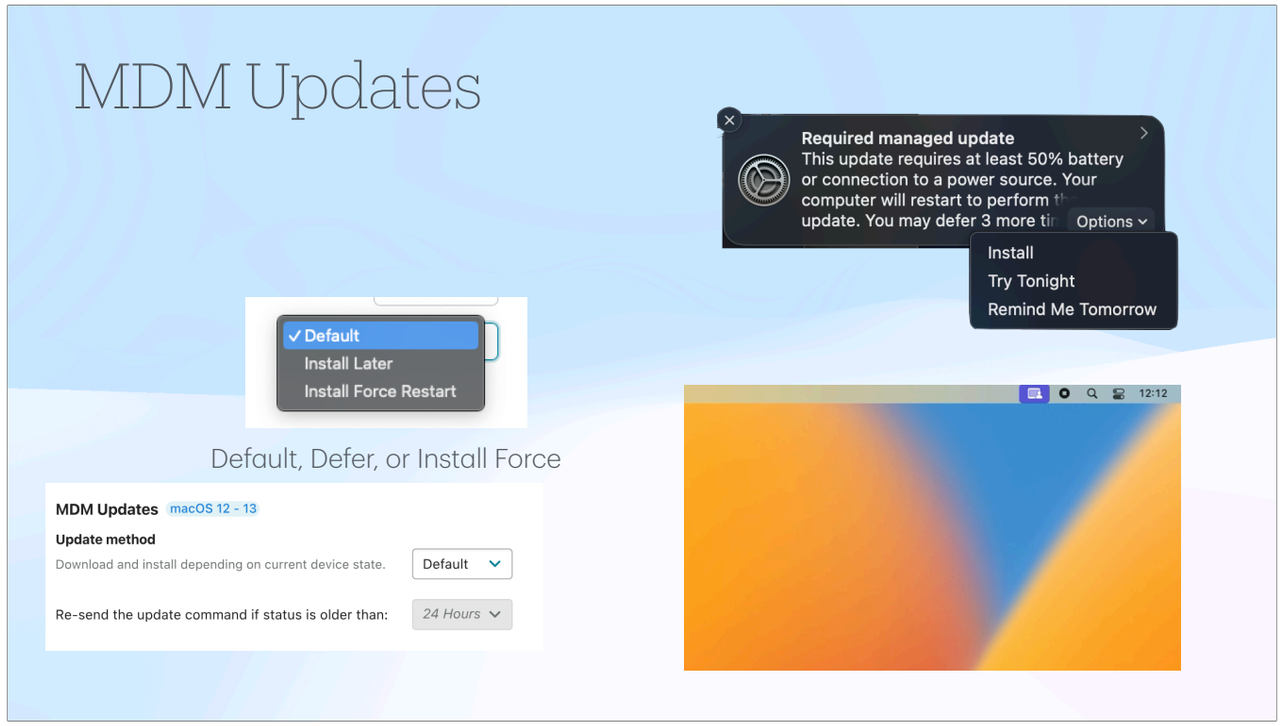
Three types of **managed** Updates and Upgrades

- MDM Updates
  - iOS 9.0+, iPadOS 9.0+, macOS 10.11+, tvOS 12.0+
- Enforcement Specific Declaration
  macOS 14+, iOS 17+, iPadOS 17+
- Software Update Settings (Global Settings Automatic Actions) Declaration
  macOS 15+, iOS 18+, iPadOS 18+

So now we have 3 types of managed Updates and Upgrades, the MDM Updates, the Enforcement Specific Declarations, and the Software Update Settings, Global Settings, Automatic Actions Declaration.

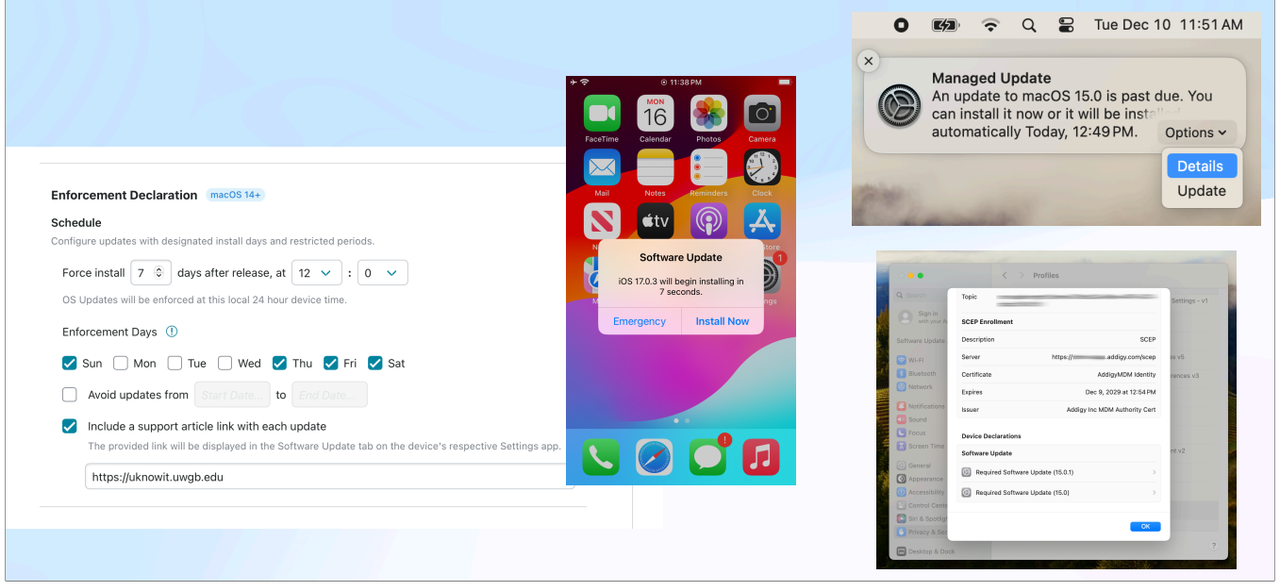
End User & Admin Experience

So with that, what does the End User & Admin experience look like for each of these?

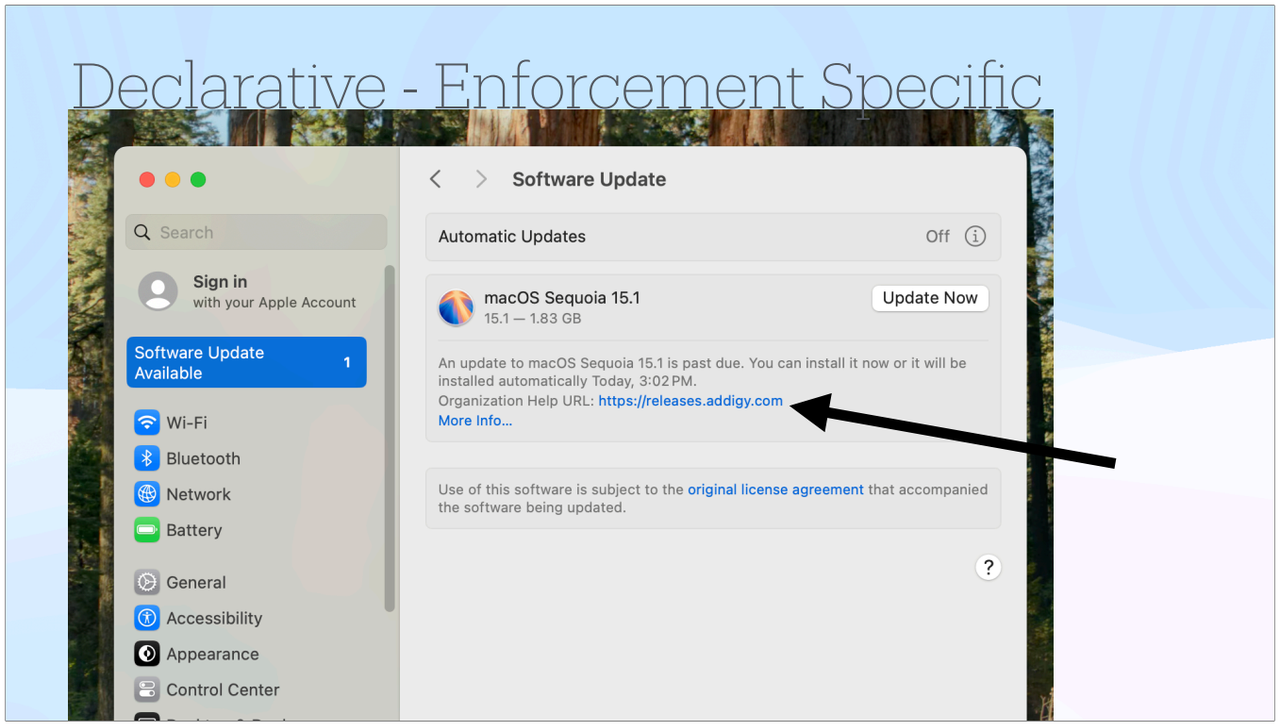


I'll go quickly through MDM updates, which again was deprecated as of WWDC and will be removed next year, likely for macOS 27 and up. You're able to send out default, install later, or install force restart to determine the device's behavior with it. If you do a default, end users get that 60 second countdown to restart once the update is prepped and ready. With install later, end users get a prompt where the admin can set a number of times it can be deferred. Then we have install force restart, which is good for those one-offs where maybe you're on a call with someone and providing support in real time.

Declarative - Enforcement Specific

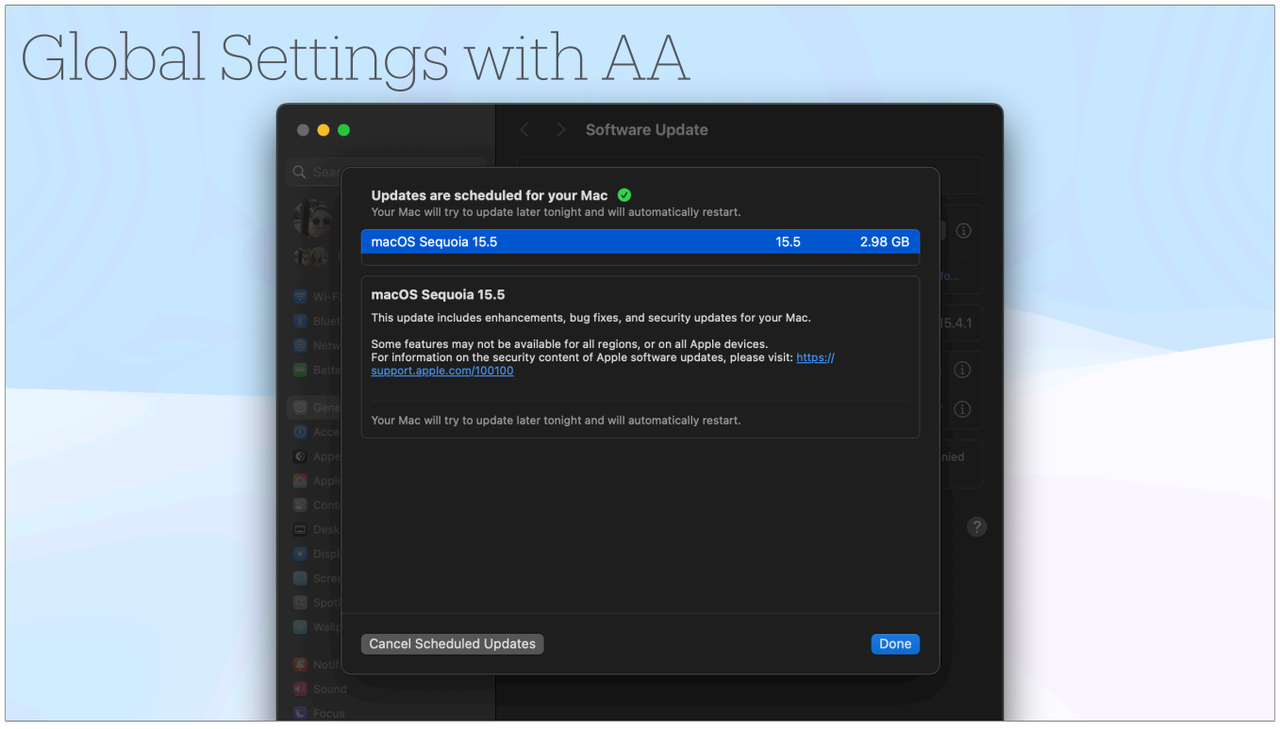


Enforcement specific varies from vendor to vendor, I have a screenshot here of Addigy because that's where I work, but within Jamf you'd pick a specific date and time to go and do it, each vendor does it a bit differently. Basically, you're saying I'm sending a specific time that the device will go on, in it's set timezone on that date, to a specified build version I want it to do. You can also include a URL that will show up in system settings if you want to provide some context to the user.

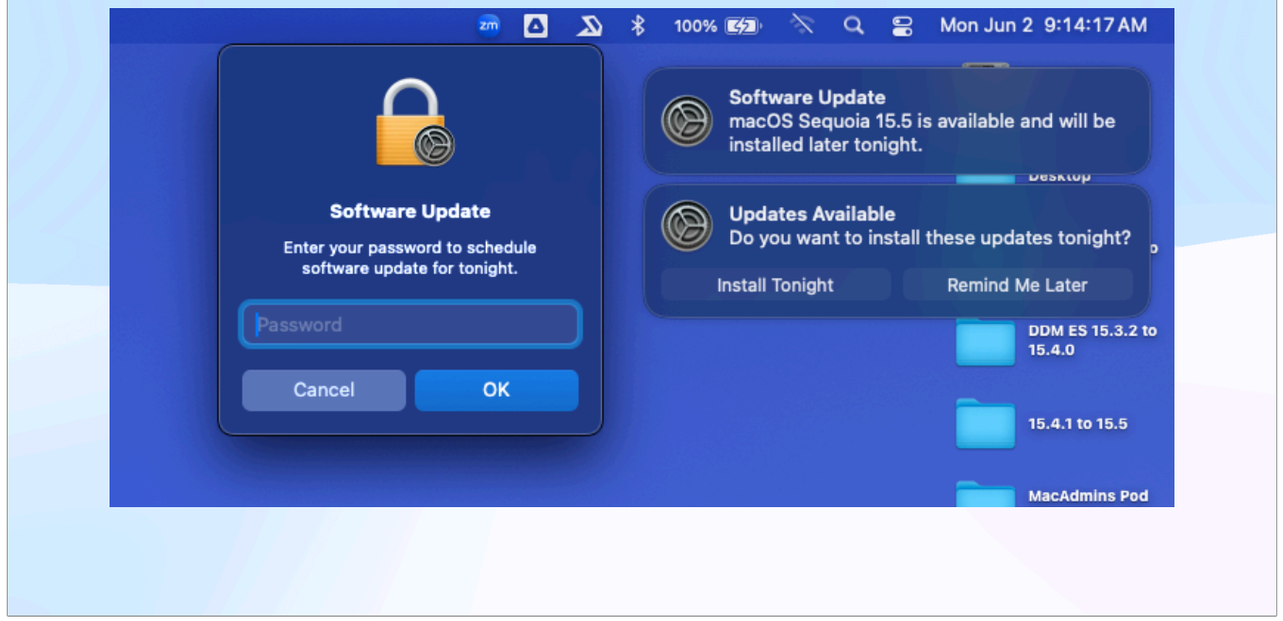


That URL shows up here for any user that is curious enough to go into System Settings. This is also where users would see the date when the update will happen automatically, or in this case they'll see that it is past that due date and will run at the specified time.

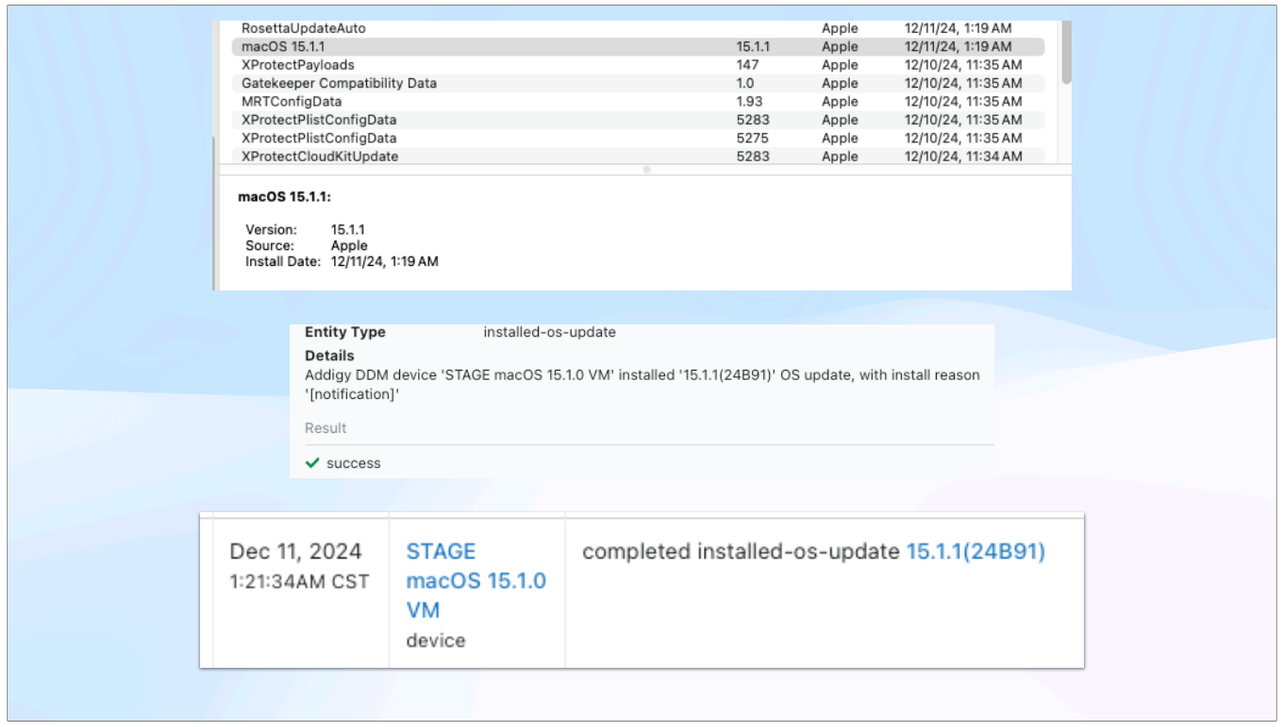
Global Settings with AA



Now the new Global Settings with Automated Actions is a little different in that it doesn’t give you a specific time, it just says ‘hey, we’re doing this later tonight’ once the update is downloaded and prepped - it uses on-device machine learning to determine what time is suitable for that. It will always try to do it overnight if the device meets the criteria to authorize that update, and we’ll talk a little about what the criteria is in a minute.



The update will try to use their last saved state of their password or passcode from the last unlock to perform that, but it may prompt the user if that doesn’t work. It will try to use the Bootstrap token from the MDM vendor, but that isn’t foolproof and there are several feedback tickets filed with Apple about it being hit or miss even when the token exists.



What does that look like if it happens automatically? These are some screenshots from a VM that was left running overnight, if we look at the install history in the system report, we can see the update went through at 1:19am on December 11th. Information on this then gets sent to your MDM Vendor to update it on the device's status.

Timelines

- MDM Updates
  - Command sent, command acknowledged, update/upgrade runs
- Enforcement Specific Declaration
  - Time to enforce (install) by via TargetLocalDateTime, TargetBuildVersion, TargetOSVersion, and DetailsURL in Settings.app
- Settings Declaration (Automatic Actions) Objects
  - Sets things not pertaining to a direct install event at takeover...
    - AllowStandardUserOSUpdates, Notifications, RecommendedCadence, RapidSecurityResponse
  - Most importantly it sets the timeline for update install
    - **AutomaticActions**, Deferrals, and Beta

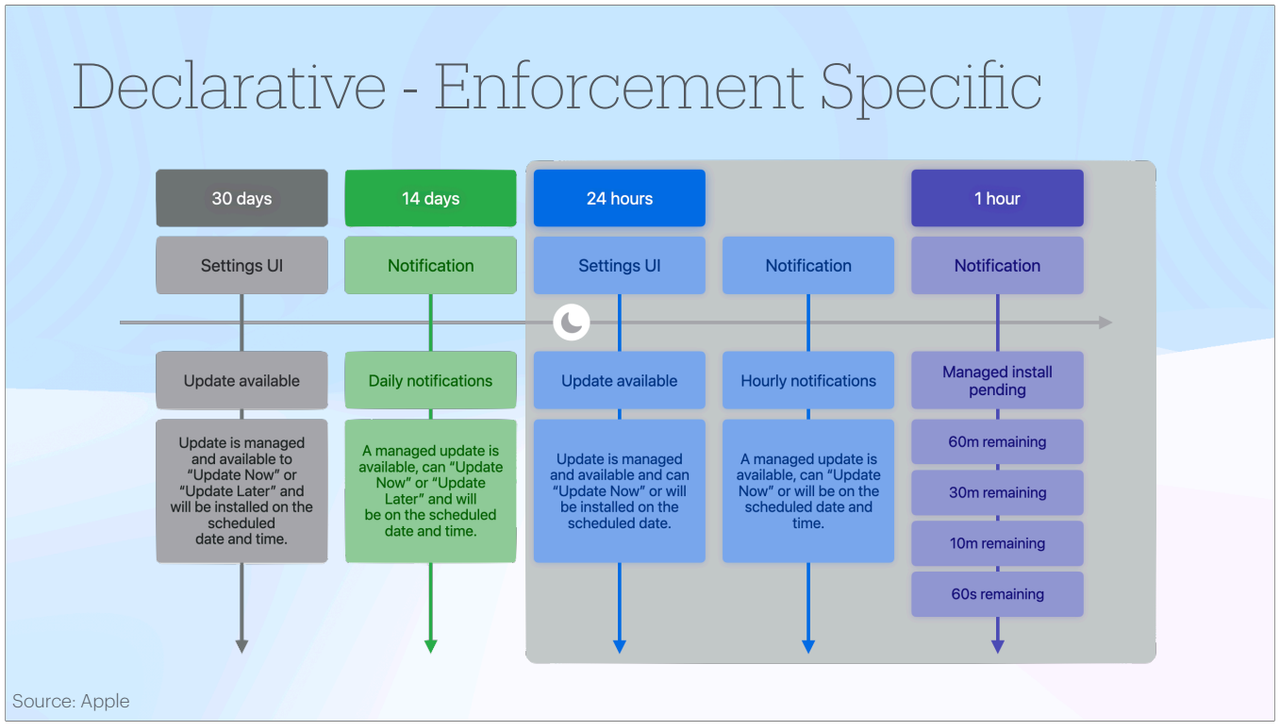
Next we'll talk about what the timelines look like for each of these, since that's important to consider when thinking about both the end user and admin experiences so that you'll know what's going to take place more specifically.

With MDM updates, you're sending that command, the command is acknowledged, and that update is going to run with whatever setting for default, install later, or force reboot that was set.

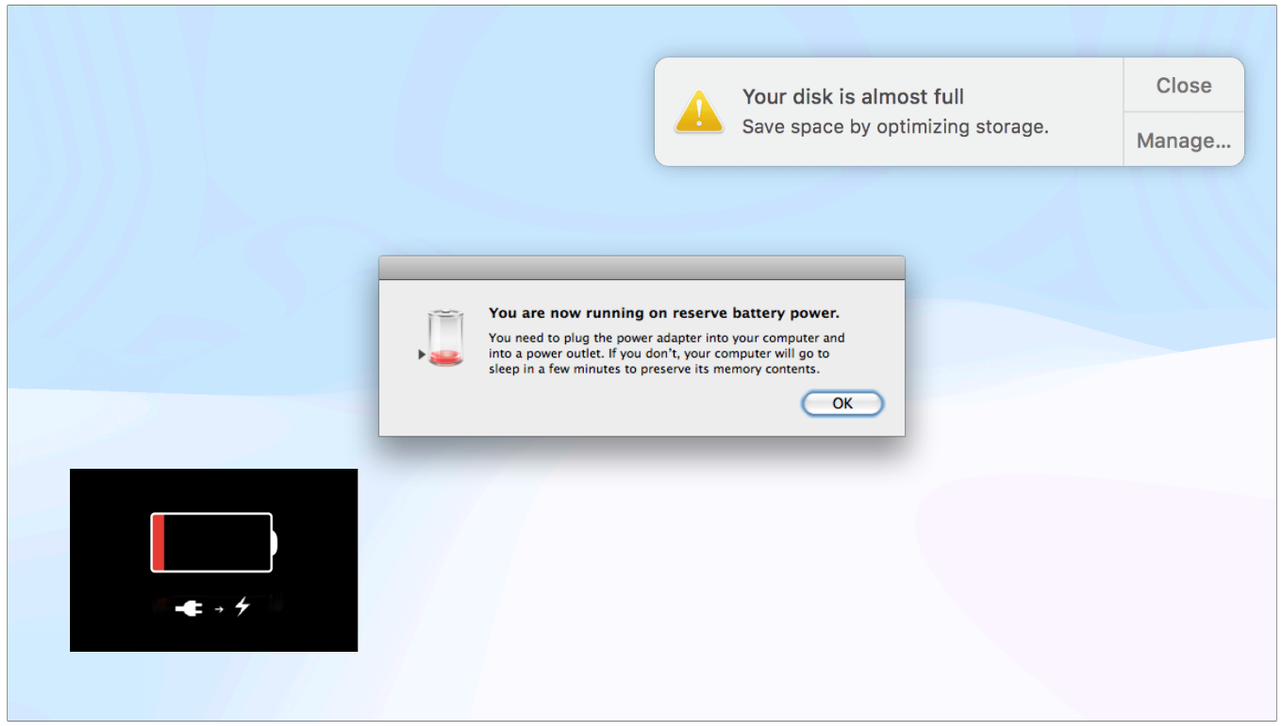
The enforcement specific declaration, you'll again be sending that target local date time, build version, OS version, and an optional details URL.

The newer stuff with the Settings Declaration with Automatic Actions kinda has 2 parts to it, it's all part of the same object but things not directly related to the install would be that I can choose if Standard users can do the updates themselves if they get a prompt, you can control the number of notifications they get, the recommended release cadence for iOS which is handled a little bit differently than macOS, and you can control that Rapid Security Response updates and if the user can install them or remove them. Most importantly though, you're setting the timeline for that install. It's turning on those Automatic Actions to download the update and install the update automatically, you're setting a deferral window of 1 to 90 days, for example if you set 7 days, it won't show the user the option to upgrade or give them notifications until the day 7, and then it will begin to notify them and follow the settings you have in place. We've seen that in production, the update usually goes through one or two days after that deferral window. You also have some Beta options available if you choose to do that.

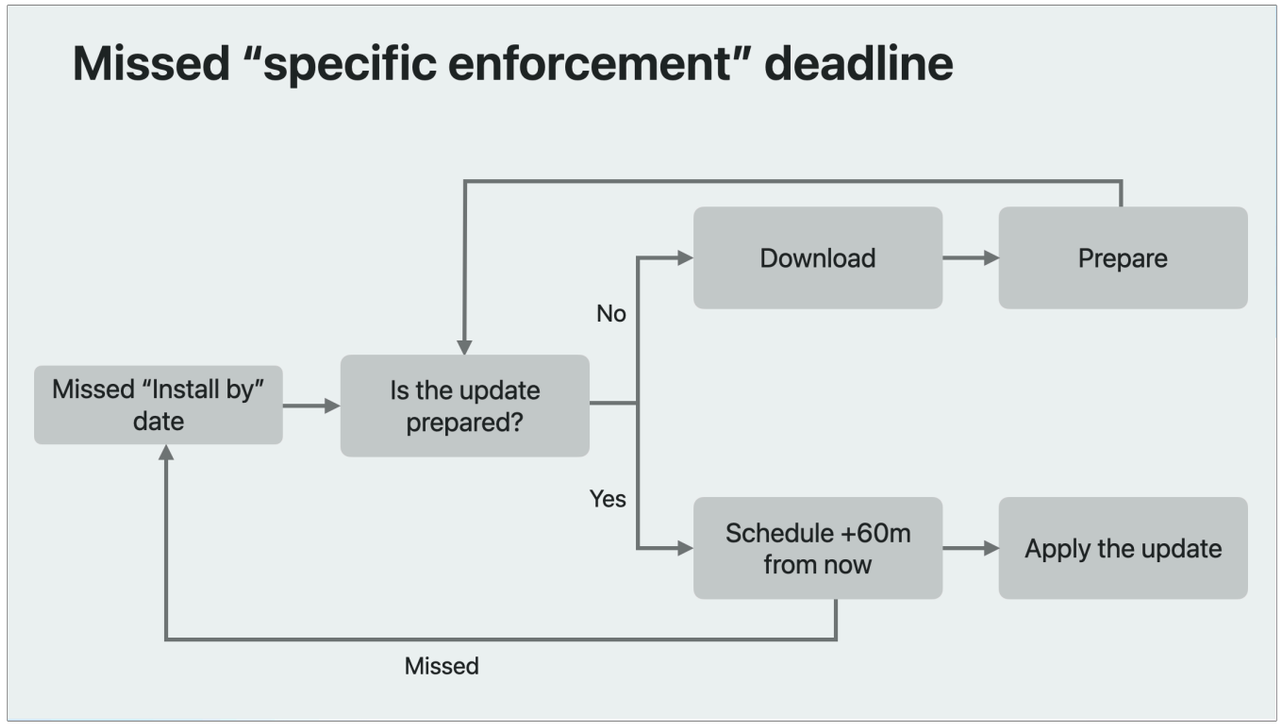
I mentioned that there is criteria for when a device will do an update. It needs a network so it can download the update, it needs a certain state of charge, and if the end user is using the device or has apps running that don’t allow for a save state and reboot, it will stop the process. Apple does have a KB on the minimum charge needed for different types of updates, generally you’ll be safe if it is 50% or more.



Let’s deep dive into an example of an Enforcement Specific Declaration. We’ve said we want to install the update at a specific date and time, 30 days from now. The end user will get a notification that it is available, they can go ahead and install it, it will not be pre-downloaded for them yet so they’ll also download it as part of the user-initiated install. At 14 days, the download has happened and the update is prepped to run, users can hit a notification to Update Now or Later. Since users will postpone in, the behavior changes at 24 hours remaining. The notifications will now bypass Do Not Disturb and the user will get hourly notifications until the last hour when it goes to 60 minutes, 30, 10, and finally 60 seconds, and based on support tickets, that 60 seconds is where the user sees the notification for the first time and when the time is up, the install and reboot is launched, even if you’re in the middle of say, a Zoom call.

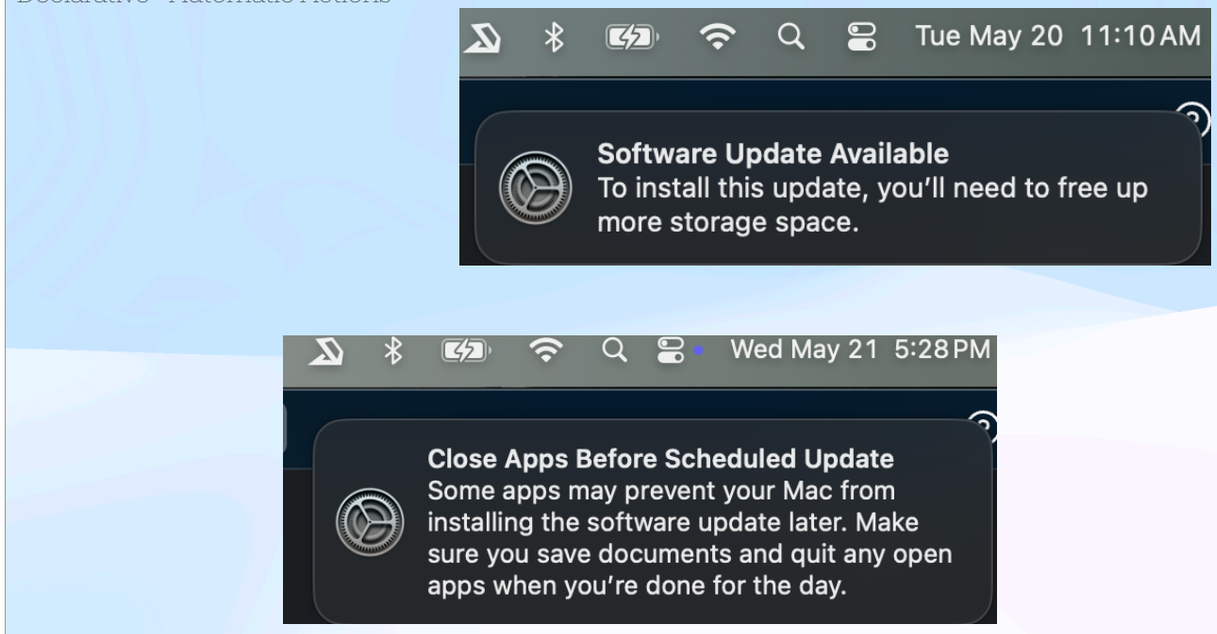


There are going to be times when devices still haven't installed the update despite the time window passing, there can be a few reasons for that - the device isn't charged enough, didn't have a network connection, or the disk was too full to download that update.



So what happens if we miss that specific enforcement deadline? Basically, it goes through this flow over and over until the update is installed.

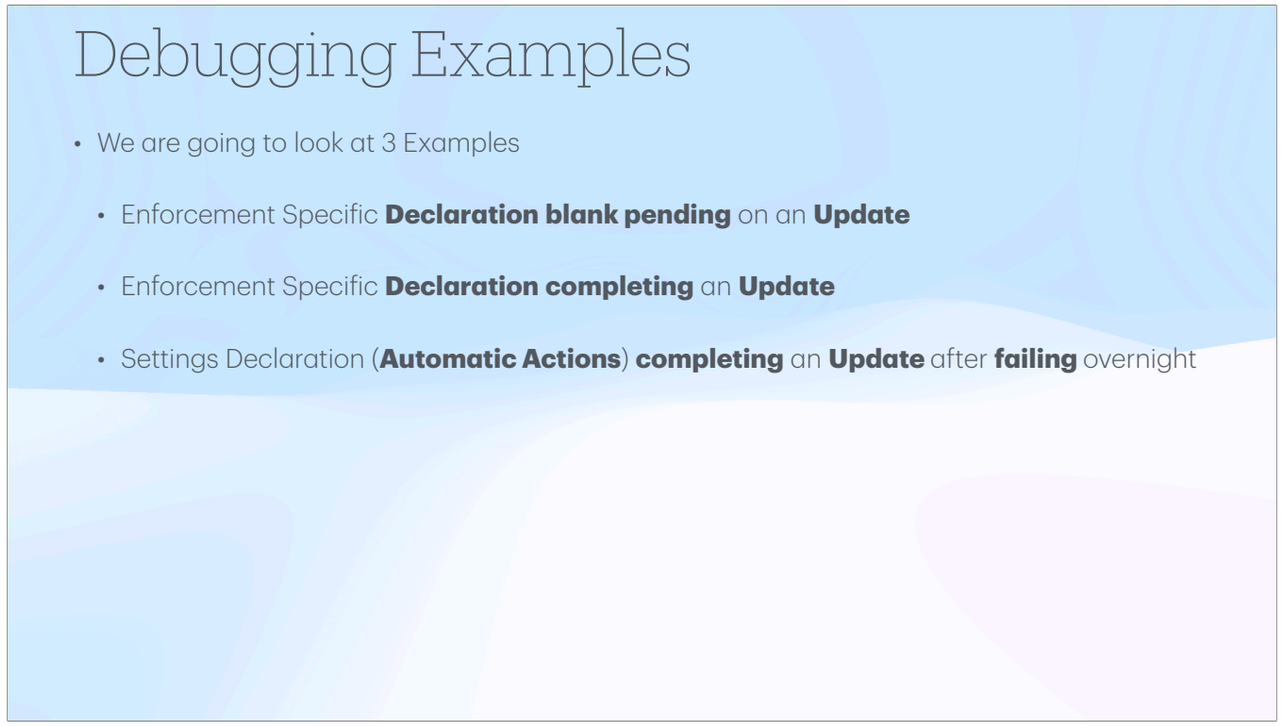
Declarative - Automatic Actions



Automated Actions are less aggressive…on-machine AI and more notifications - admin has less control, but generally a better end user experience

Debugging

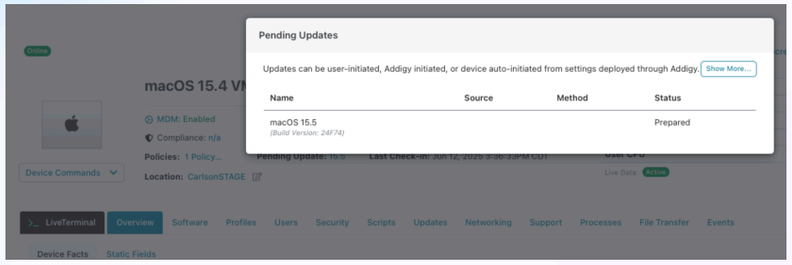
Time to really dive in with a few examples of debugging, everybody’s favorite part when working with OS Updates.



We'll take a look at 3 examples today. I'm going through declarative examples in the interest of time today, but if you did want to see an MDM Upgrade example check out the video on Youtube of Bryce doing a similar talk at Penn State MacSysAdmin.

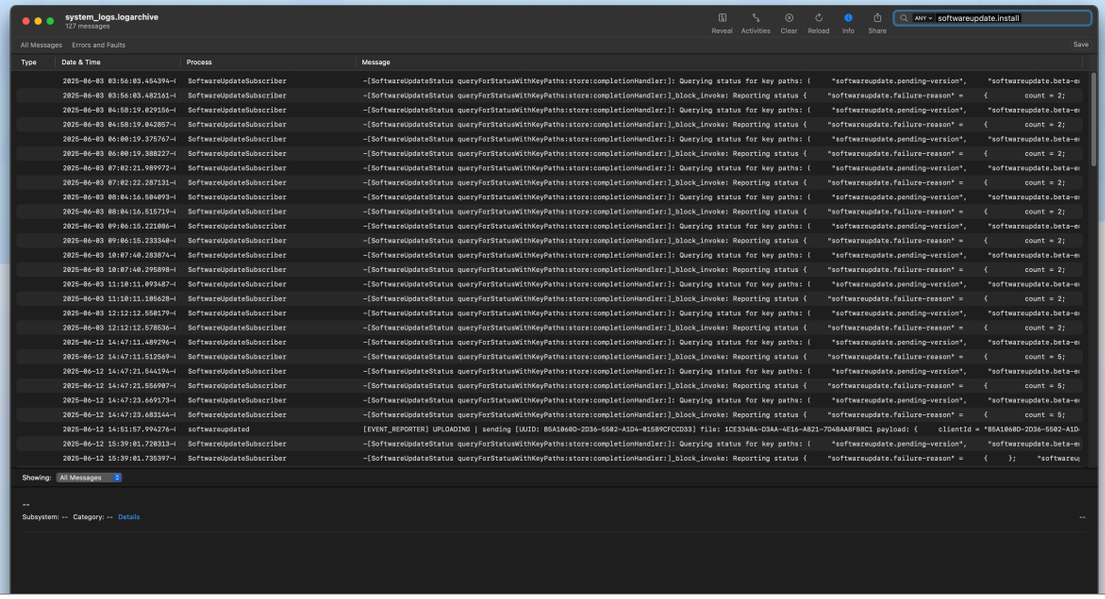
DDM Enforcement Specific - Blank Pending

- macOS 15.4 to 15.5
- Pending Enforcement Specific for a future time but the status channel was not reporting anything as to where it was at
- VirtualBuddy VM on wired network



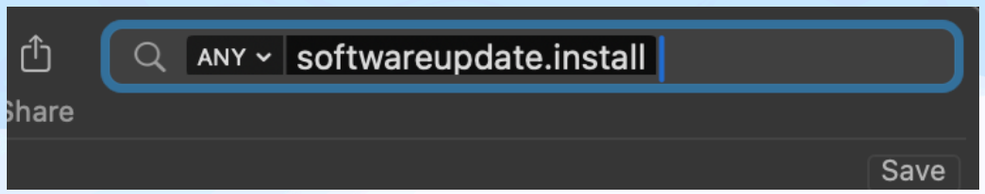
First let’s take a look at an example of an enforcement specific update that is stuck pending, but there isn’t really any information on why it is pending. This example is for macOS 15.4 to 15.5 with a pending update for a future enforcement time. The status channel was reporting that there was something, but it wasn’t really telling me what it was actually doing. The screenshot there is looking at the device in more detail in Addigy, and if we click on Pending Updates the status is prepared, but we really aren’t getting more information than that here.

DDM Enforcement Specific - Blank Pending



That means it is time to hop into the logs for it.

DDM Enforcement Specific - Blank Pending



Then we filtered by software update.install - you could get more granular than this, but this is a good place to start when investigating declarative updates, especially for Enforcement Specific declarations because this basically gives you a timeline based on whatever the status channel would be reporting, so you can just go down to see what is happening and where it gets held up. Then you can rip off that filter and figure out what was happening at that point in time. Start with this, find your timestamp, and then work backwards from there.

DDM Enforcement Specific - Blank Pending

default 2025-06-03 03:56:03.482161 -0500 **SoftwareUpdateSubscriber** -[**SoftwareUpdateStatus** queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"**softwareupdate.failure-reason**" = {

count = 2;

reason = "Error Domain=SUMacControllerError Code=7507 \"[SUMacControllerErrorAccessRequestDenied=7507] Access request was denied\" UserInfo={NSDebugDescription=[SUMacControllerErrorAccessRequestDenied=7507] Access request was denied, NSLocalizedDescription=**The software update request for this process was denied as another process is currently performing an operation. Please try again later.**}";

timestamp = "2025-05-31T14:51:36-05:00"; };

"softwareupdate.install-reason" = {

reason = (

); };

"softwareupdate.install-state" = prepared;

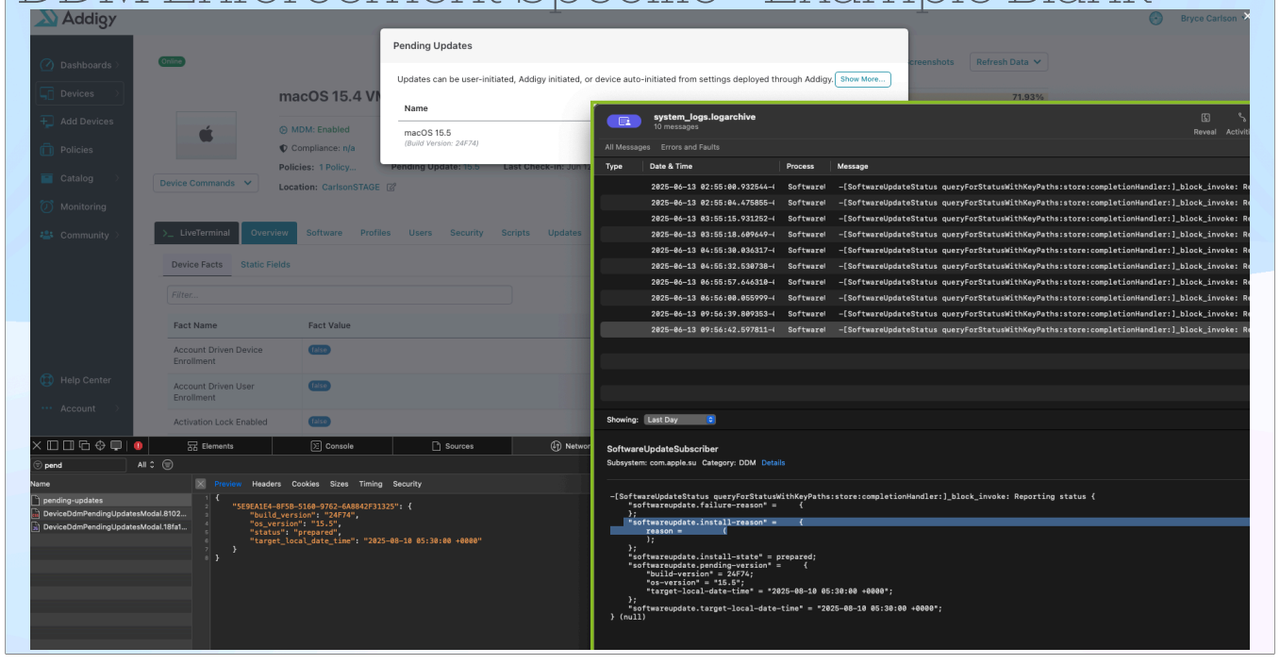
"softwareupdate.pending-version" = {

"build-version" = 24F74;

"os-version" = "15.5";};} (null)

Looking at our example when we filter by it, we see software update. Failure reason. So this is one of the components within the status channel report. Looking at it we can see the reason was the process was denied by another process that is performing an operation. Please try again. Then we see the install-reason there is blank, because there is no install reason. We can also see the .install-state is prepared and that the os version is 15.5, so we know it downloaded it and tried to unpack it, but it can’t go beyond that point

DDM Enforcement Specific - Example Blank



So why is it showing as blank in the UI, well we weren't getting the full picture of the information because the install-reason was blank. We've made a code change so the information gets pulled in now, but this is an example of where the troubleshooting may just be because you don't have as much information in your MDM vendor UI as you do in the logs.

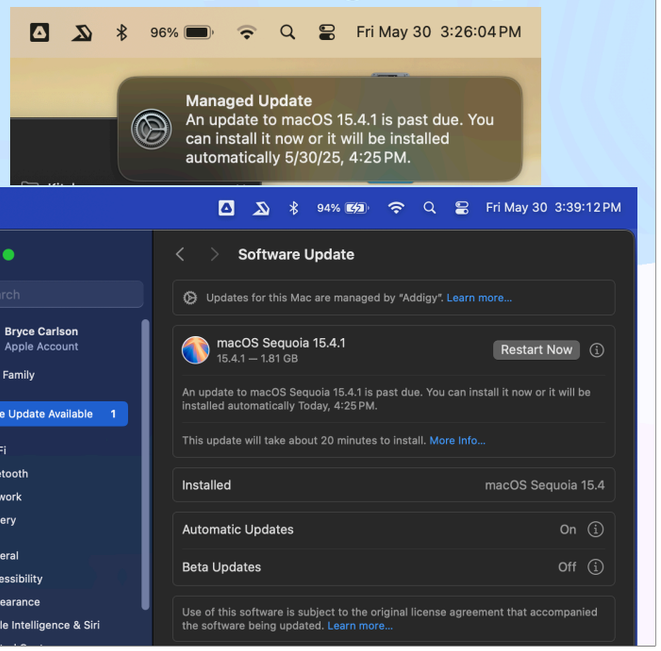
softwareupdate.install-reason

- system-settings
  - It was triggered from Settings (on iOS and iPadOS) or System Settings (on macOS).
- Install-tonight
  - It was triggered by an install tonight action.
- auto-update
  - It was triggered by an automatic update.
- notification
  - It was triggered from a user notification.
- Setup-assistant
  - It was triggered during Setup Assistant.
- command-line
  - It was triggered by the softwareupdate command-line tool.
- mdm
  - It was triggered by an MDM command.
- declaration
  - It was triggered by a declarative device management configuration **enforcement specific**

In our example the install-reason was blank, but here are some examples of things you could see if you’re filtering by software update.install

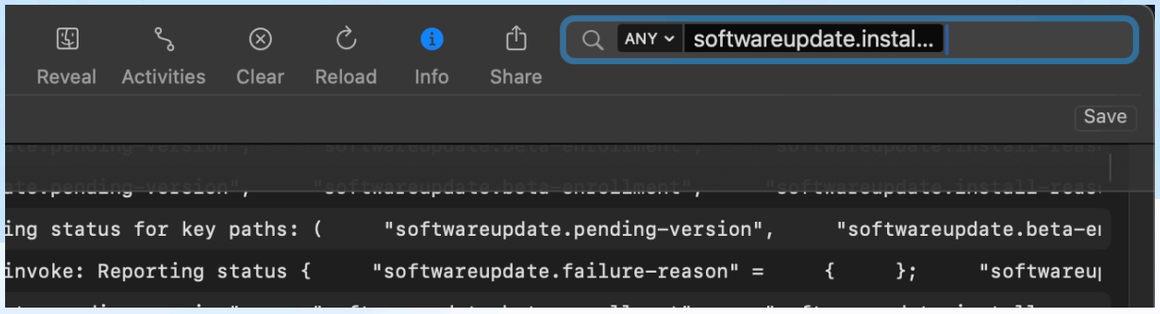
DDM Enforcement Specific - Completing an Update

- macOS 15.4 to 15.4.1
- Pending Enforcement Specific for a **past due time**
- Silicon M1 MacBook Pro
- My live production Addigy Mac



I shamelessly got these screenshots from Bryce, but this example is going to be taking us from macOS 15.4 to 15.4.1 and we are past due for the set enforcement time, and the device is currently in Pending.

DDM Enforcement Specific - Completing an Update



“softwareupdate.install-reason”

Now if we hop into console and take a look at the logs and look at install reason.

DDM Enforcement Specific - Completing an Update

```
default2025-05-30 15:19:42.903564 -0500  SoftwareUpdateSubscriber   -[SoftwareUpdateStatus
queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {
    "softwareupdate.failure-reason" =     {
        count = 0;
    };
    "softwareupdate.install-reason" =     {
        reason =         (
            notification
        );
    };
    "softwareupdate.install-state" = none;
    "softwareupdate.pending-version" =     {
    };
} (null)
```

We can see that initially that before the device got the declaration of past due, there was no failure reason, no install reason, and no install state, and no pending version.

DDM Enforcement Specific - Completing an Update

default 2025-05-30 15:25:39.558770 -0500 **SoftwareUpdateSubscriber SUOSUDDMController**: **Scheduling update with declaration: SUCoreDDMDeclaration**
(DeclarationKey:com.apple.RemoteManagement.SoftwareUpdateExtension/0D6549DE-4D91-4037-B086-17CEC6934A89:MzI4M2M0OWQtMTE3NC00MWEwLWJlMzEtNGY5MzJiMmVjYmNkIC0tIDE1LjQuMSAtLSAyMDI1LTA1LTMwIDAwOjAwOjAw.MjdlYjRkNDhkMjkzZjc1ODRmZDcxZDQ4Y2Q3ZmNhN2Q=|
EnforcedInstallDate:2025-05-30T00:00:00|VersionString:15.4.1|BuildVersionString:(null)|DetailsURL:(null)|companyName:(null)|NotificationsEnabled:YES)

Then we see the update subscriber DDM controller getting a scheduled declaration, and if we look at the bottom 3 lines we can see that it sends down the enforced install date, which in this case was past due, it includes the version number, and we've got notifications enabled for this declaration.

DDM Enforcement Specific - Completing an Update

default2025-05-30 15:25:48.362248 -0500 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.**failure-reason" = {};**

"softwareupdate.**install-reason**" = {

reason = (

declaration

);};

"softwareupdate.install-state" = none;

"softwareupdate.**pending-version**" = {

"build-version" = 24E263;

"os-version" = "**15.4.1**";

"target-local-date-time" = "2025-05-30 21:25:47 +0000";

};

So the next thing that takes place then is if we go back to that update status, we'll see that failure reason is blank because it hasn't failed at anything yet, the install reason is marked as declaration, the install state is none right now because it hasn't actually started doing the install. And the pending version is 15.4.1 with the declaration that we have there.

DDM Enforcement Specific - Completing an Update

default2025-05-30 15:25:57.498830 -0500 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.failure-reason" = { };

"softwareupdate.install-reason" = {

reason = (

declaration

);};

"softwareupdate.install-state" = downloading;

"softwareupdate.pending-version" = {

"build-version" = 24E263;

"os-version" = "15.4.1";

"target-local-date-time" = "2025-05-30 21:25:47 +0000";

};

So after that it starts downloading, and if an end user were to look in System Settings they would see the download status being updated in the Software Update panel.

DDM Enforcement Specific - Completing an Update

default2025-05-30 15:37:23.972968 -0500 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.failure-reason" = { };

"softwareupdate.install-reason" = {

reason = (

declaration

);};

"softwareupdate.install-state" = prepared;

"softwareupdate.pending-version" = {

"build-version" = 24E263;

"os-version" = "15.4.1";

"target-local-date-time" = "2025-05-30 21:25:47 +0000";

};

After a little bit of time, the download should complete and the install state will shift to prepared.

DDM Enforcement Specific - Completing an Update

default2025-05-30 16:26:12.348973 -0500 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.failure-reason" = { };

"softwareupdate.install-reason" = {

reason = (

declaration

);

};

"softwareupdate.install-state" = installing;

"softwareupdate.pending-version" = {

"build-version" = 24E263;

"os-version" = "15.4.1";

};

Then the install state will shift from prepared to installing. The device goes through the upgrade, does its thing.

DDM Enforcement Specific - Completing an Update

default2025-05-30 **16:29:07.627366** -0500 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"**softwareupdate.failure-reason**" = {

count = 0; };

"softwareupdate.install-reason" = {

reason = (

);

};

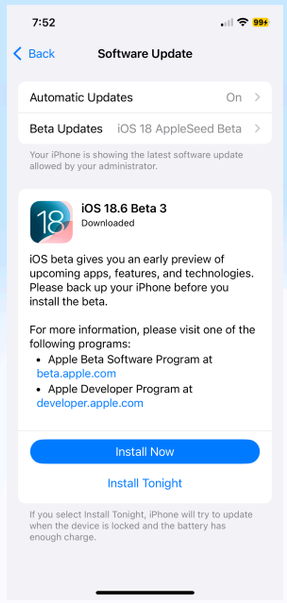
"softwareupdate.install-state" = none;

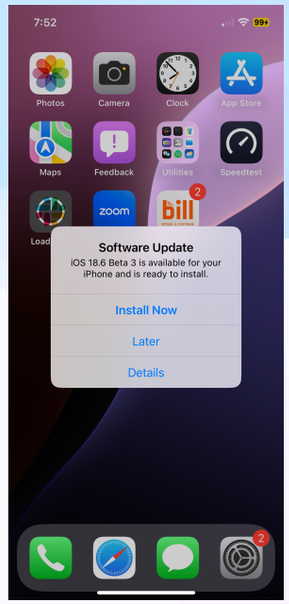
"softwareupdate.pending-version" = { };

Then it’s done. The upgrade completed, it doesn’t have a failure reason. It no longer has an install-reason or state, because it is now in the version that the declaration says it needs to be on. And because of that, the log file doesn’t have much to look at, which is good.

Settings Declaration (Automatic Actions) - iOS

- iOS 18.6 B3
- AppleSeed for IT Beta
- 1 Day Deferral
- Auto Actions Enabled
- iPhone 12





On to our last example of the day, which will be on iOS - the macOS process is very similar, but I thought we should add in an iOS example for good measure. Because of how automatic actions has been architected, both the software update status and the status channel are very uniform across the platforms.

For this example we're going to go to iOS 18.6 Beta 3, it is part of the AppleSeed for IT Beta, a 1 day deferral has been set, and automatic actions have been enabled.

Settings Declaration (Automatic Actions) - iOS

default2025-07-16 00:45:01.081168 -0400SoftwareUpdateSubscriber -[**SoftwareUpdateStatus** queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.beta-enrollment" = "iOS 18 AppleSeed Beta";

"softwareupdate.failure-reason" = {

count = 0;

};

"softwareupdate.install-reason" = {

reason = (

);

};

"softwareupdate.install-state" = none;

"softwareupdate.pending-version" = {

};

} (null)

So the first thing we see when the device gets the declaration is that we want it to be in the AppleSeed Beta and we don’t have an install state yet because there’s no version yet until it goes out and does a query of what versions are there.

Settings Declaration (Automatic Actions) - iOS

default2025-07-16 11:45:38.745002 -0400 SoftwareUpdateSubscriber -[**SoftwareUpdateStatus** queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.beta-enrollment" = "iOS 18 AppleSeed Beta";

"softwareupdate.failure-reason" = {

count = 0;

};

"softwareupdate.install-reason" = {

reason = ( );

};

"softwareupdate.install-state" = prepared;

"softwareupdate.pending-version" = {

"build-version" = **22G5073b**;

"os-version" = "18.6";

};

Once it does that query, it says ‘hey there’s a version’ and it went out and downloaded it, so now we see an OS version of 18.6 with a build version ending in 73b that is in the prepared install state. Now we’re going to let the example device sit on a desk or night stand for a while, plugged in, but not being used.

Settings Declaration (Automatic Actions) - iOS

default 2025-07-16 11:45:38.898298 -0400 SoftwareUpdateSubscriber -[**SoftwareUpdateStatus** queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.beta-enrollment" = "iOS 18 AppleSeed Beta";

"softwareupdate.failure-reason" = {

count = 1;

reason = "Error Domain=com.apple.softwareupdateservices.errors Code=20 \"InstallationKeybagRequired, PasscodeLocked\" UserInfo={NSDebugDescription=InstallationKeybagRequired, PasscodeLocked, SUInstallationConstraintsUnmet=48, SUMDMInstallationRequest=false}";

timestamp = "2025-07-16T11:45:38-04:00";

};

"softwareupdate.install-reason" = {

reason = (.); };

"softwareupdate.install-state" = prepared;

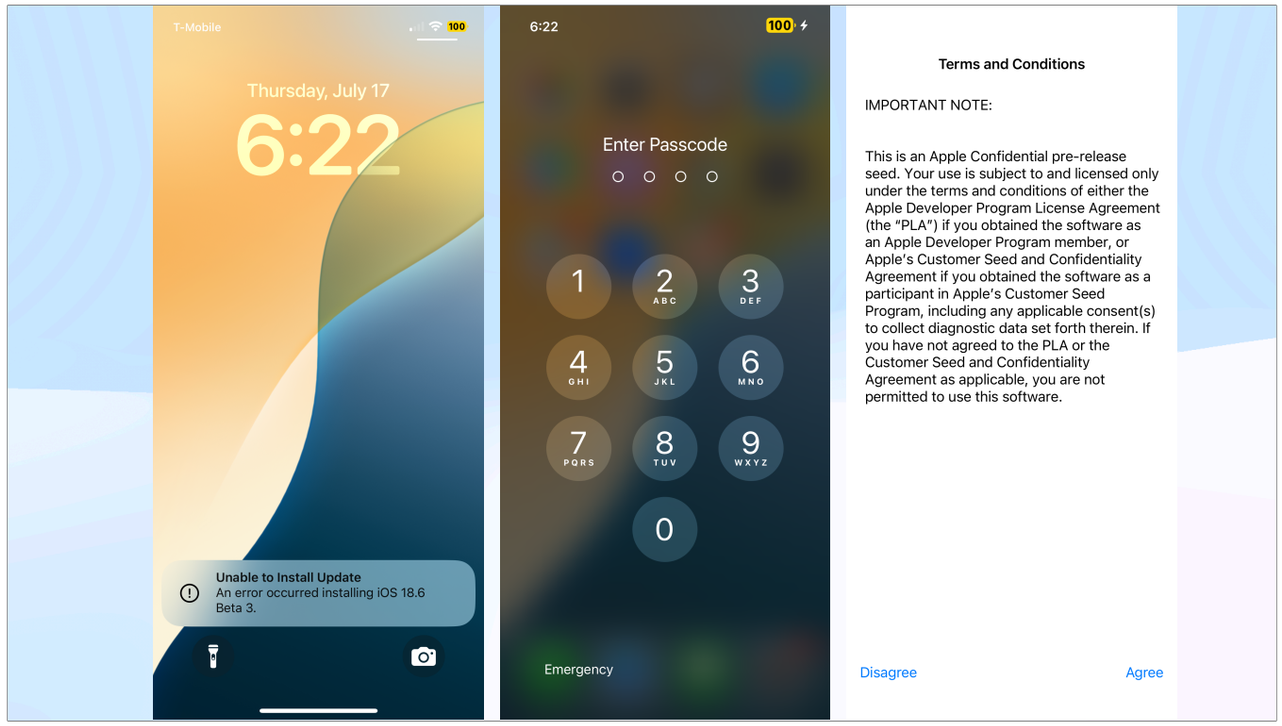
"softwareupdate.pending-version" = {

"build-version" = 22G5073b;

"os-version" = "18.6";

};

Now in this case, that example device doesn't update. Weird. It tries again, and again. We're getting a failure reason for it though, and it's a code 20 that the Installation Keybag is required and we have a Passcode Locked message in there. So what happened is that after the device was last rebooted, we didn't actually authenticate in with that PIN code, so it is just sitting there plugged in, waiting to update, but it can't because the device is in still in a locked state.



Eventually, we’ll come back to the device and there is a message on the home screen that the update failed to install. We’ll then enter our passcode, and in this case get a pop up for Terms and Conditions. Now in this example, we actually aren’t sure if it was just the authentication blocking the update or the authentication and Terms and Conditions, since the only log failure we get is related to the passcode.

Once the device is unlocked and the terms and conditions are agreed to though, that update kicks off and begins the verification and install process.

default 2025-07-17 **06:25:01**.396563 -0400 SoftwareUpdateSubscriber -[SoftwareUpdateStatus queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting status {

"softwareupdate.beta-enrollment" = "iOS 18 AppleSeed Beta";

"softwareupdate.failure-reason" = {

count = 0;

};

"softwareupdate.install-reason" = {

reason = ( ); };

"softwareupdate.install-state" = prepared;

"softwareupdate.pending-version" = {

"build-version" = 22G5073b;

"os-version" = "18.6";

};

If we go back in to the logs, we'll see that the failure-reason is gone and the install-state has flipped to prepared.

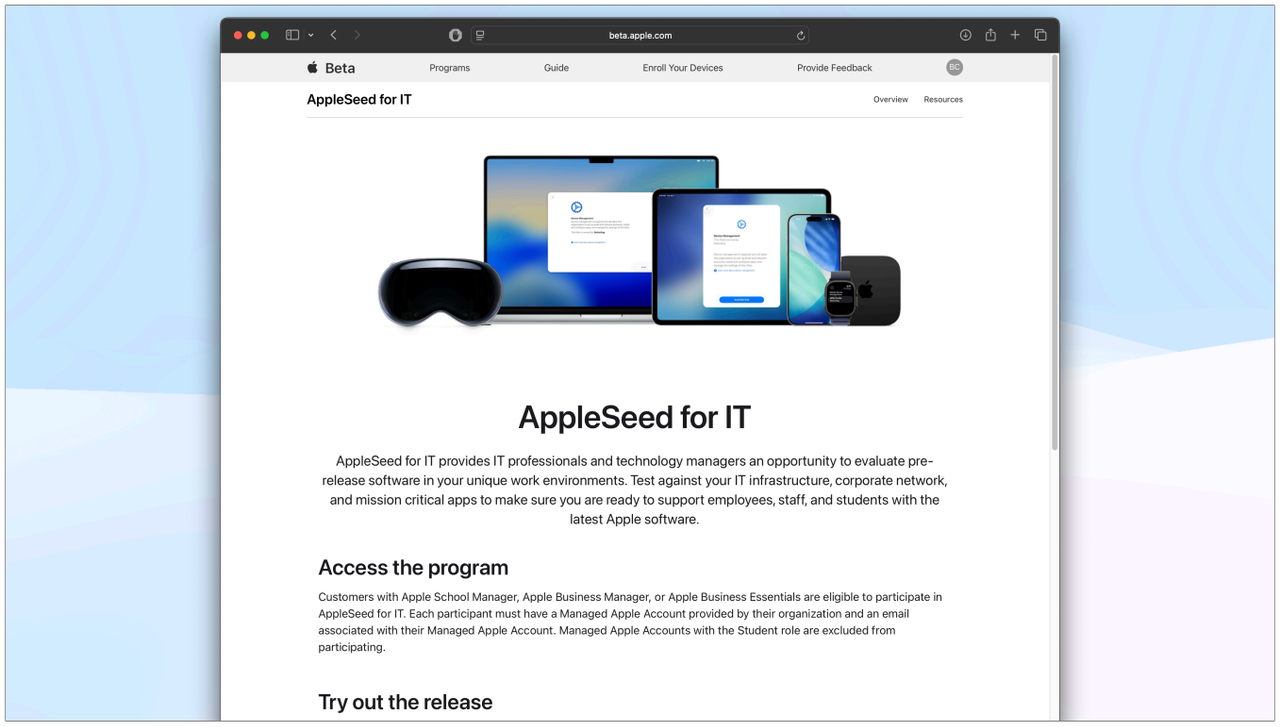
```
default2025-07-17 06:29:30.735569 -0400   SoftwareUpdateSubscriber   -
[SoftwareUpdateStatus
queryForStatusWithKeyPaths:store:completionHandler:]_block_invoke: Reporting
status {
    "softwareupdate.beta-enrollment" = "iOS 18 AppleSeed Beta";
    "softwareupdate.failure-reason" =     {
        count = 0;
    };
    "softwareupdate.install-reason" =     {
        reason =         (
        );
    };
    "softwareupdate.install-state" = none;
    "softwareupdate.pending-version" =     {
    };
```

And after just a little bit more time, that log will empty out because there is no update that is needed, it worked and that means there is nothing to see here.

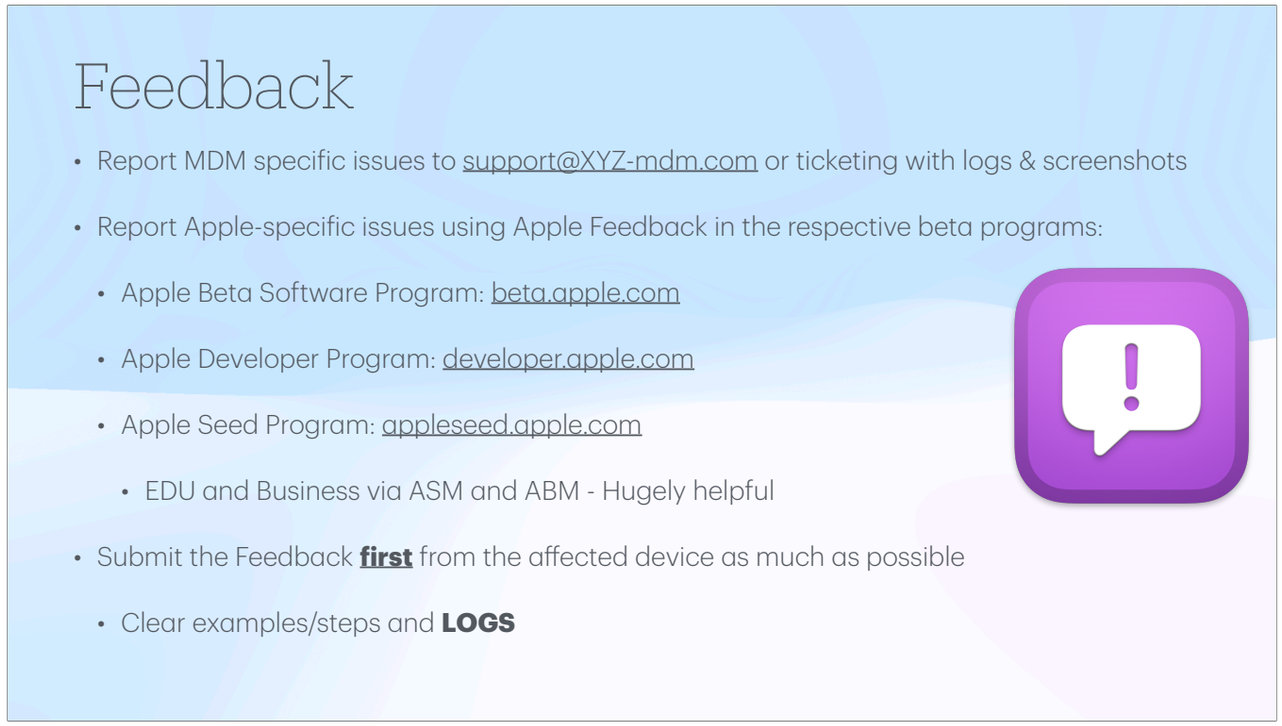
Providing Feedback

Hopefully those examples were helpful and gave you some ideas on where to start when debugging Declarations.

To wrap things up I'll go over some different ways you can help provide feedback to both Apple and your MDM vendors.



One of the most important things in influencing the future of OS updates is the Apple Seed program. If you’re not in it, get in it. If you’re in Apple School Manager, business manager, or business essentials, you are eligible to participate in any of the beta programs. You can do updates using the settings declaration to test things out, and if you enroll in Apple Seed, Apple gives you a lot of helpful resources such as documentation and config profiles for testing along with the betas. It’s great to get feedback in while the new OS is still in development and in that beta phase, because in my experience things are more likely to get changed then rather than if you file feedback later.



If you end up coming across something you have feedback on, there are a few ways to file it. If it's something MDM specific, you'll want to contact your MDM vendor's support at whatever their email address is and include logs, screenshots, some other helpful things to include are a description of expected behavior vs actual behavior, and steps to reproduce what you're seeing.

Now, if it's in Apple-specific feedback, you'll want to make sure you are going to the appropriate place for whatever state you are in, so if it's in a beta, file feedback through the beta software program. If it's in the Apple Seed, file it through Apple Seed. One thing to stress, is submit the feedback if at all possible from the affected device first. This is because of the way tagging happens on the Apple side, it makes the whole process much smoother if you're able to first submit feedback from the affected device or VM. Not always possible, I know, but when it is definitely make sure to do so.

Thank You!