OLM V1 MVP

OLM overview and use cases

OLMs purpose is to manage cluster extensions centrally and declaratively on Kubernetes clusters. Its mission is to make installing, running, and updating functional add-ons to the cluster easy, safe, and reproducible for cluster administrators and PaaS administrators throughout the lifecycle of the underlying cluster as well as those of add-ons. The scope of OLM is currently the one cluster it is running on.

OLM has unique support for the specific needs of cluster extensions, which are commonly referred to as operators. These are classified as one or more Kubernetes controllers shipping with one or more API extensions (CustomResourceDefinitions) to provide additional functionality to the cluster (though there are deviations from this coupling of CRDs and controllers, discussed below). They are managed centrally by OLM running on the cluster, where OLMs functionality is implemented following the Kubernetes operator pattern as well.

OLM defines a lifecycle for these extensions in which they get installed, potentially causing other extensions to be installed as well, a limited set of customization of configuration at runtime, an update model following a path defined by the extension developer, and eventual decommission and removal.

There is a dependency model in which extensions can rely on each other for required services that are out of scope of the primary purpose of the extension, allowing each extension to focus on a specific purpose. OLM also prevents conflicting extensions from running on the cluster, either with conflicting dependency constraints or conflicts in ownership of services provided via APIs.

Since cluster extensions need to be supported with an enterprise-grade product lifecycle the specific scenarios in which an extension can be installed or updated are going to be limited by the author of the extension, primarily to align with what was tested by their QE processes. OLM allows the author to enforce these support limitations in the form of additional constraints.

During their lifecycle on the cluster, OLM also manages the permissions and capabilities extensions have on the cluster as well as the permission and access tenants on the cluster have to the extensions. This is done using the Kubernetes RBAC system in combination with tenant isolation using Kubernetes namespaces. The interaction surface of the extensions is solely composed of Kubernetes APIs the extensions define.

OLM also defines a packaging model in which catalogs of extensions, usually containing the entire version history of each extension, are made available to clusters for cluster users to browse and select from. Via new versions of extensions delivered with this packaging system, OLM is able to apply updates to existing running extensions on the cluster in a way where the integrity of the cluster is maintained and constraints and dependencies are kept satisfied.

The scope of all these features is a single cluster so far with namespace-based handling of catalog access and extension API accessibility and discoverability. Expansion of this scope is indirectly expected through the work of the Kubernetes Control Plane (KCP) project, which in its first incarnation will likely use its own synchronization mechanism to get OLM-managed extensions deployed eventually on one or more physical clusters from a shared, virtual control plane called a “workspace”. This is an area under active development and subject to change, OLM might need to become aware of KCP in a future state.

Verdict: The purpose of OLM remains the same. The scope of OLM will increase to span multiple clusters following the KCP model, though likely many aspects of this will become transparent to OLM itself through the workspace abstraction that KCP provides. What needs to change in OLM 1.0 is how the above tasks are carried out from the user perspective and how much control users have in the process and which persona is involved.

Functional Requirements

Priority Rating: 1 highest, 2 medium, 3 lower (ex. P2 = Medium Priority)

F1 - Extension catalogs (P1): The existing OLM concepts around catalogs, packages and channels is to be used as a basis for below functional requirements.

F2 - Extension catalog discovery (P2): Unprivileged tenants need to have the ability to discover extensions that are available to install. In particular users need to be able to discover all versions in all channels that an extension package defines in a catalog. The privilege should be given at the discretion of the cluster administrator.

F3 - Dependency preview (P3): Before extension installation, OLM needs to introspect the dependencies of an extension and present a preview of the resolution result to the user to let the user confirm they are ok with this.

F4 - Dependency review (P3): For installed extensions OLM needs to surface the dependency relationship to other installed extensions and also highlight which other extensions depend on a particular extension so users are informed about the relationships at runtime.

F5 - Permission preview (P2): Before extension installation and updates, OLM needs to allow introspection of the permissions the extension requires on cluster and dependencies APIs. This is especially important during updates when the permission scope increases in which case updates should be blocked until approved (F14).

F6 - Install/Update preflight checks (P1): When installing and updating extensions OLM should carry out a set of preflight checks to prevent installations from going into a failed state as a result of attempting an upgrade or install. Preflight checks should include availability and (if applicable) health of (running) dependencies and any cluster runtime constraints (F19).

F7 - Extension installation (P1): OLM needs a declarative way to install extensions either from a catalog of extensions or directly from an OCI image. Should the installation attempt fail due to unfilled requirements, constraints, preflight checks or dependencies there needs to be an option to force the install. Extensions are cluster-wide singletons, thus they can only be installed once in the cluster and are managed at cluster-scope.

F8 - Semver-based update policy (P2): OLM should allow users to specify if and when extensions are updated. Manual update policy should include the user explicitly approving an update to be installed. An automatic update policy should allow updates to automatically be applied as soon as they are available and should provide further conditions upon which an update is to be applied. Conditions concern version changes of the extension, specifically: automatic updates on z-streams only, automatic updates on y-streams only, always automatic update. Updates across channels are outside of the update policy.

F9 - Update notification (P3): As updates can be made available at any time using OLMs existing over-the-air catalog update capabilities, OLM should provide events / notifications on the platform to notify users about available but not yet approved updates of existing installed extensions, specifically so that graphical consoles can pick them up and visualize them. Automatically applied updates as per F8 should also create notifications.

F10 - Extension updates (P2): As extensions get updated, either automatically or manually, OLM replaces the older version of the extensions with a newer version atomically. Up until any custom code or conversion logic runs, an update should be able to be rolled back (F23). When multiple extensions are updated to satisfy an update request, the update policy of each extension needs to be respected to allow users to pin certain extensions to installed versions or certain types of updates (e.g. z-stream only). It should also be possible to force an update to a certain version, even if there is no direct path as per the graph metadata. Otherwise all versions on

F11 - Request / Approval Flow for Installs / Updates (P2): To support multi-tenant environments a request / approval flow is desirable for generally available content within default catalogs. In this model any tenant with enough privilege can discover installable content and can trigger an install request, which can in turn be approved or denied by a more privileged administrative role. Such requests should also have timeouts. Administrators should have the ability to define a list of extensions that are automatically approved at the scope of a namespace. Administrators should be able to get aware of unapproved requested via alerts triggered by the platform.

F12 - Installed extension discovery (P1): Unprivileged tenants need to be able to discover installed extensions if they are offering services for them to use in their namespace. OLM needs to provide distinct controls for installed operators which administrators can use to regulate in which namespaces extensions are discoverable by tenants, irrespective of the namespaces in which the operator has permissions on cluster APIs (see F13)

F13 - Extension permissions management (P1): Administrative personas need to be able to configure in which namespaces in the cluster the extension can get the requested permissions the extension author deems required. The control needs to be independent of the controls in F12. Extensions should always be given permissions to update their own APIs (if they define any) to inform users about potential lack of permissions in their namespace.

F14 - Extension permissions escalation checks (P2): If, in the course of an update, an extension requests more permissions than the currently installed version, an automatic update is blocked and an administrative persona needs to specifically approve the update by default. The user who installed the extension can opt out of these permission increase checks for the purpose of automation.

F15 - Selective extension permissions grants (P3): Administrative personas can choose to give extensions only a subset of the permissions it requests. This should be manageable at a per namespace level.

F16 - Extension removal (P1): Administrative personas need to be able to remove an extension including all the content that was part of the original installation bundle. Special handling should be implemented for CRDs, which when not removed, are left behind in a functioning state (i.e.any dependencies on running controllers like conversion webhooks need to be removed). When they are to be removed this can only happen if the user opts into F17. Special care also needs to be taken to allow the extension to perform any clean upon getting a signal to be removed. Components need to be removed in an order that allows the extension to handle a graceful shutdown.

F17 - Extension cascading removal (P2): OLM needs to be able to cleanly remove an extension entirely, which means deleting CRDs and other resources on the cluster. In particular this means the removal of all custom resource instances of the CRDs to be deleted and all extensions with a hard dependency. A user needs to actively opt-in to this behavior and OLM has to sequence the successful removal of all affected components and the extension itself.

F18 - Standalone extension bundle installation (P2): For local development OLM should allow the installation of an extension by directly referring to the bundle / OCI image in a container registry rather than a package name of an operator in a catalog containing the image in order to simplify testing and delivering hotfixes.

F19 - Extension constraints (P1): OLM needs to allow extensions to specify constraints towards aspects of the running cluster and other currently installed or future extensions. Aspects of the running cluster should include software version, resource utilization, overall resource availability and state of configuration settings. These constraints need to be evaluated when extensions are attempted to be installed or updated.

F20 - Extension health (P1): OLM needs to be able to report the overall health state of the extension on a cluster along the following set of aspects: presence of all required objects from the extension bundle, health of all components that have a liveness / readiness endpoint, presence and health of all other extensions the extension in question has a dependency on as well as evaluation of all additional constraints from F19. An extension that was forced to install despite missing / unhealthy dependencies and violated constraints has a reduced health scope down to the liveness / readiness endpoint.

F21 - Custom extension health (P3): OLM should provide a way for extensions to report an aggregate health state with custom logic. This should align with other communications channels that are also used for extensions to declare update readiness (F25). This way extensions can report health more accurately than what OLM reports today based on simple readiness of the extension controller pod. Clients like graphical consoles should be able to make use of this to supply additional overall health states of extensions that provide some form of control plane by the user of other extensions.

F22 - Best effort resolution (P2): OLM should always try its best to resolve installation and update requests with the currently available and healthy set catalogs to resolve against. Intermittently or permanently failed catalogs should not block resolution for installation and updates. Fulfilling user requests is valued higher than determinism of results.

F23 - Opt-in to fallback / rollback (P2): OLM should allow operator developers to specify whether or not it is safe to rollback from a particular current version of the operator to an author-specified previous version, once an extension update has passed pre-flights checks in F10 but subsequently failed to become available or carry out a migration. In these cases OLM should allow the administrator to downgrade the operator to the specific previous version. OLM should also respect this downgrade path when conducting updates that fail and use it to fail back to the previous version of the operator indicating that the downgrade path is supported. Extension uptime is an important goal of OLM.

F24 - Extension Overrides (P2): Components deployed as part of extensions will require user-provided modifications at runtime in order to aid features like placement, networking configuration, proxy configuration etc. that require customization of volume mounts, annotations, labels, environment variables, taints, tolerations, node selectors and resources. OLM should support accepting these customizations to the extension as input from the user prior or after the installation of an extension and apply them to applicable objects such as Deployments, StatefulSets, ReplicatSets.

F25 - Extension-controlled Update Readiness (P2): Extensions should be able to control their readiness for updates. An extension could be on a critical path or in a state where updates would lead to disruption or worst-case: outages. OLM should respect these update readiness signals and allow the extension to signal readiness differentiated to what nature the update is based on semantic versioning rules, i.e. patch updates vs. minor or major updates. Once the signal is encountered, OLM should block the update until the signal disappears.

F26 - Canary Style Rollouts (P3): OLM should have an opinion about how administrators can carry out roll outs of new extension versions that coexist with already installed versions of the extension, in particular if the extension only ships a controller. While conflicting CRDs cannot co-exist, controllers that only selectively reconcile objects (Ingress operator pattern) can. OLM should support these deployment styles while ensuring integrity of cluster-level extensions like CRDs.

F27 - Pluggable certificate management (P2): OLM should rely on other controllers to create and lifecycle TLS certificates required to drive the functionality of certain extensions, like webhooks that need to trust / need to be trusted by the cluster's API server. OLM should not implement any certificate handling itself. In a first implementation support should be established for cert-manager.

Behavioral Requirements

Priority Rating: 1 highest, 2 medium, 3 lower (ex. P2 = Medium Priority)

B1 - Single API control surface (P1): While the underlying implementation of the functional requirements can be carried out by different controllers with different APIs, to the administrative and non-administrative users there should be a single, cluster-level API that represents an installed extension with all high level controls described in / required by F4, F7, F8, F10, F13, F14, F15, F16, F17, F18, F21, F22, F23 and F24.

B2 - GitOps-friendly API surface (P1): In many cases OLM APIs will not be used by a human via a CLI or GUI interactively but declaratively through a GitOps pipeline. This means the primary OLM API to lifecycle an extension cannot leak abstractions to other APIs for initial deployment or reconfiguration. Modifications must not require conditional lookup or modifications of other objects on the cluster that are created as part of the declarative intent stored in git in the form of YAML manifest files.

B3 - Declarative API (P1): As an operator itself, OLMs API controls have to allow for operations to be carried out solely declaratively. This mandates continuous reconciliation and eventual consistency of desired state. OLM should not conduct one-off operations. OLM should not require either clean up of failed operations or restating intent to retry a failed operation (with the exception of F11).

B4 - Event trail (P2): OLM should make heavy use of Kubernetes events to leave an audit trail of tasks and actions carried out on the cluster. For expected failure scenarios administrators should not need to consult the OLM controller logs for debugging but solely rely on events and status conditions (see also B6).

B5 - Force overrides (P1): While OLM has a lot of opinions about safe operations with cluster extensions they do not apply all the time since OLM cannot possibly foresee how extensions behave at runtime. It needs to yield to the user in cases where they have more certainty about what's going to happen based on their background knowledge of the cluster or the operator. It should offer ways to force-override decisions that OLM

made that block the user from proceeding in a certain direction, especially in the areas of extension installation, removal and updates.

B6 - Human-readable status extensions information (P2): Whenever OLM is in the process of or having reached or failed to reach a desired state it needs to update the user about what is happening / what has happened without assuming knowledge about OLM internal or implementation details.

B7 - Scalability & Resource consumption (P1): OLM is used on clusters with hundreds to thousands of namespaces and tenants. Its API controls, specifically for F2 and F12 need to be built in such a way that resource consumption scales linearly with usage and cluster size and the overall resource usage envelope stays within manageable bounds that does not put the cluster stability, especially that of the API server at risk. System memory especially is a scarce resource.

Compatibility Requirements:

C1 - Compatibility with existing extensions (P1): OLM should be able to manage extensions packaged with the current bundle format in the way described by the functional and behavior requirements when the bundle supports AllNamespace installation mode.

C2 - Compatibility with existing catalogs (P1): OLM should be able to retrieve and update extensions that adhere to C1 from the currently supported catalog formats (File-based catalogs).

C3 - Incompatibility with existing extensions (P1): OLM 1.0 does not support managing bundles or extension versions that do not support AllNamespace installation mode with the new set of APIs or flows

Assumptions
No additional tenancy model will be introduced at the control plane / API layer of Kubernetes upstream

KCP doesn’t fundamentally change OLMs role and responsibilities around managing extensions (at least initially)

OLM will move to a descoped, cluster-wide singleton model for cluster extensions, extension management isn’t namespaced

Constraints
Only operator bundles with “AllNamespace” mode installation support can be lifecycled with the new APIs / flows in OLM 1.0

Dependencies

RBAC templating (OLM-2158) and discrete visibility control (OLM-2366) will land prior to OLM 1.0 and unblock most operators that do not support AllNamespace installation mode today

Bandwidth in the Console engineering team to surface the new flows / forms / visualizations associated with any functional requirements

Migration

A new set of APIs is introduced in parallel to the existing set of APIs

Users opt-in to the new set of APIs, potentially resulting in a reinstall of their extension if required

Extensions that are shipped with the current bundle format with AllNamespace mode can simply be reused with the new set of APIs and controls

Extensions that do not support AllNamespace mode cannot be managed with the new APIs

Migration scripting is provided to mass-convert existing installed extensions (“Subscription” / “OperatorGroup” objects) on existing clusters to the new OLM 1.0 model assuming they are compatible

Operator authors that are also SRE/Managed PaaS administrators are incentivized to make their operator compatible with the requirements of OLM 1.0 to reap the operational benefits

TODO

- Definition of "extension"
- Does OLM become ELM? Does this provide of provisioning bundles that do not add APIs?