

RFC 850 **WIP** ▾: Improving The Upgrade Experience - Upgrades API

Author:

Date: 2023-01-01

Status: Review ▾

Decider: Nelson Araujo

Input providers: JH Chabran Michael Lin Jacob Pleiness Louis Jarvis

Approvers (please review by EOD YYYY-MM-DD):

Approvals:

Team ([follow these conventions](#) so we can search by team):

Updated: 3/26/2024 to reflect new knowledge and changes in plans around CRD use

Summary

Upgrading Sourcegraph is often a tedious process for admins who are often expecting button click upgrades. With the acceptance of [RFC 792: Consolidation of distribution channels](#) we'll get closer than ever the north star, button click upgrades. **We want click op upgrades.** This RFC proposes some of the new application architecture that will create an API around migrator such that the service will be callable via some user interface. This was initially intended for the appliance deployment type and now more generally for k8s type deployments.

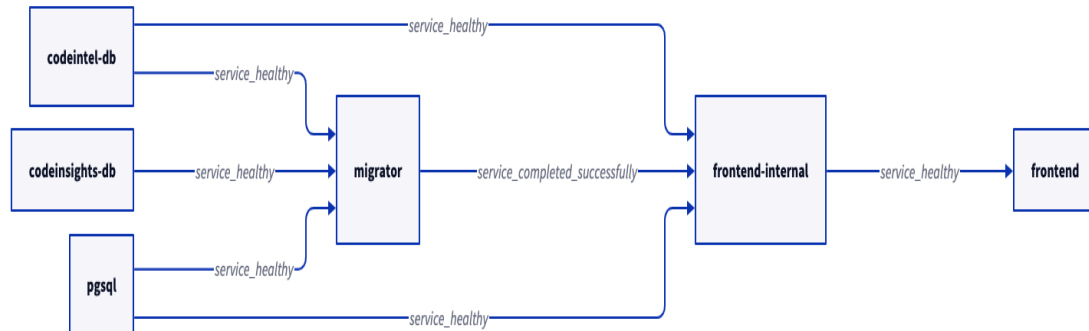
Background

There are a few key concepts about historic upgrades to keep in mind with regards to this RFC:

- The primary commands of migrator for managing the database schema are:
 - **Up**: This is the default entry command of migrator and is always run before frontend starts. This is triggered via `initContainer` in k8s, or a `depends_on` clause in docker-compose. **Up** runs all migrations defined between sequential minor versions (including patch migrations). It also acts as a validation for the frontend ensuring the expected migrations have been run before startup.
 - **Upgrade**: This is run manually by an admin and is used to migrate a Sourcegraph instance across multiple minor versions. It checks for database drift before running migrations. *This command **does not** run migrations defined in patches, and must be followed up with **up**.*

- **Migrator** is written like a cli tool and has registered commands. In our deployment repos it is run as a kubernetes job or short lived docker container, exiting when its entry command operations are completed.
- **Autoupgrade** things to be aware of:
 - In v5.0.0 and later a column was added to the **versions** table in the **frontend** database, this value is set by an admin via the site admin *updates* UI, and is used to trigger the [autoupgrade](#) flow. Autoupgrades are conducted by the **frontend** service rather than **migrator**.
 - During upgrades, and *especially multiversion upgrades*, other services should not be making requests to the database. We've advised that a multiversion upgrade requires downtime, and that all services besides the databases are brought down during a multiversion upgrade. **Autoupgrade** gets around this by starting a separate proxy frontend displaying the status of the upgrade, and [setting the dsn](#) configuration provided by the frontend to prevent other services from reaching the dbs during upgrades.

Current Startup Model: Docker-Compose



The diagram above depicts the startup pattern of a Docker-Compose deployment. Here we see that the databases are started first and that migrator must complete before other processes are started. **Migrator** is started with the **up** command. This ensures that the databases are in the expected state before frontend will start. In kubernetes deployments the same startup behavior is accomplished by running migrator as an **initContainer** of the **sourcegraph-frontend** deployment with the **up** command.

*There is a great deal of context necessary to understand the workings of our release process, its relationship to the **migrator** service, and how this ultimately influences our upgrade*

process. Ultimately this is out of scope for this RFC though. If you'd like more background check out: [v5.2 Upgrade/Migrator Design](#)

Problem

Currently the upgrade process requires git ops, and interaction with `kubectl` or `docker` manually by an admin. The Appliance deployment will allow us to simplify the upgrade process, but the current cli tool based model of `migrator` and the on startup behavior of `autoupgrade` don't fit well into the appliance architecture. Here are a few reasons why:

- **Separation of concerns:** In `autoupgrade` upgrades the frontend service is responsible for running migrations and presenting an alternate UI. The frontend already has too many responsibilities. In the appliance model we want the `operator` to be the only service to create or close containers.
- **Fallback UI:** In current Sourcegraph deployments when the application fails users are generally presented with a 503 response and must identify the source of the issue with Sourcegraph using their container orchestration tooling. If `migrator` detects an issue with the instance's dbs on startup it will fail and exit, often causing a crash loop with little other signal. This is a pain for self hosted customers.
- **Unpredictable Startup Behavior:** With `autoupgrade` upgrades are triggered on startup and contain the logic to distinguish between a multiversion upgrade and a standard upgrade. When an `autoupgrade` is started by the frontend, a [poison pill DSN](#) is set in the database connection configuration our services reference to connect to our DBs. In effect counting on failure in other services while the migrations are conducted by the first frontend to [acquire a lock](#) on the `migration_logs` table. While this approach is working, it still requires customers to understand a lot of state.

Proposal

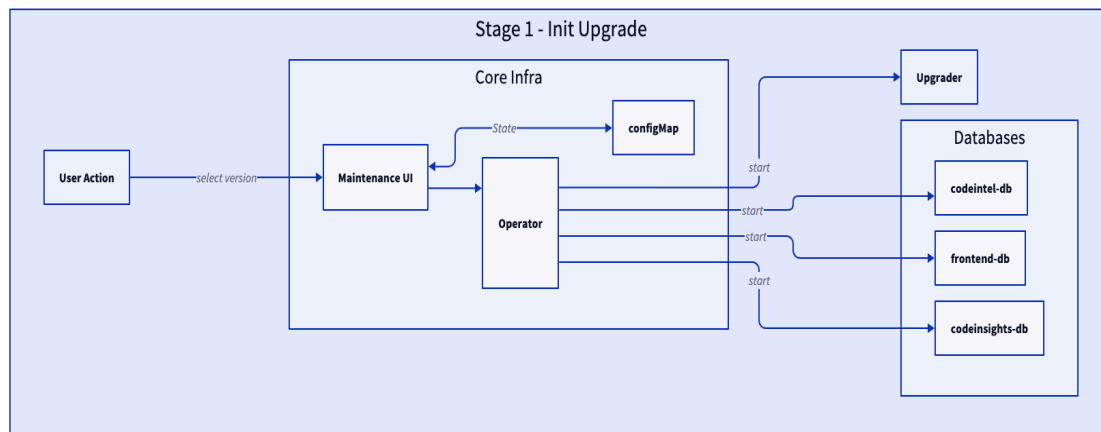
With the appliance model maintenance page, we can display useful information during times in which the frontend is down. Without changing `migrator` we want a service that knows how to communicate with the maintenance UI during upgrades. To do this we'll create a light weight service serving as an API interface for the `migrator` cli commands.

This proposal introduces some of the architecture that will enable the upgrade experience shown in this prototype:

 [YouTube: Appliance End-To-End Easy Install and Upgrades](#)

Appliance Upgrade Architecture In Stages

The appliance model involves much more orchestration so I'll depict it in stages. The first stage starts when the application is first started on the customer's k8s cluster, and the user selects a version to start.



Lets introduce the new services here:

- **Core Infrastructure**

This grouping represents the services that are always up and serve as something like the control plane for the Appliance. They should have no dependencies to other services and act as the entry point of the Appliance on the k8s cluster.

- **Operator**

This service is responsible for starting and stopping other services and nothing else. It will receive requests to start and stop other services. It takes the place of a human operator altering manifests and interacting with the kubernetes control plane.

- **Maintenance UI**

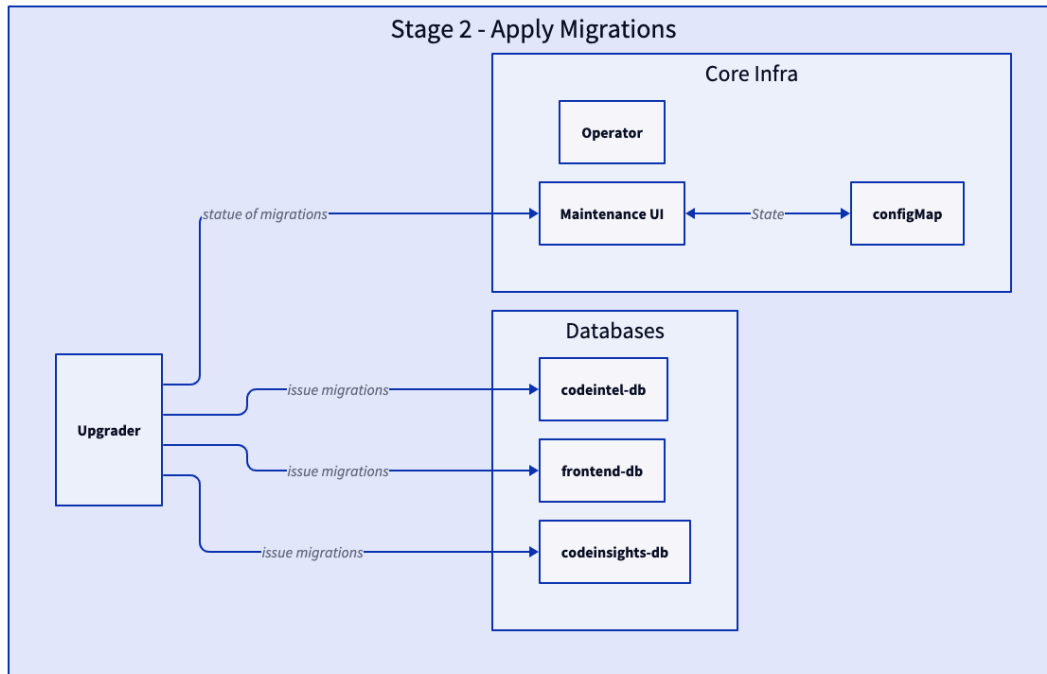
The maintenance UI is always on when the rest of the Sourcegraph application is down. This is the page displayed when `sourcegraph-frontend` can't be reached. In the case of a fresh installation or upgrade the user can select the version they'd like to install here. *With an instance thats already been initialized they'll also be able to select the version in the Sourcegraph Admin UI.*

The maintenance UI will also be capable of displaying information from other services particularly an installer/upgrader. This should roughly have parity with whats available in the autoupgrade UI of our other deployments.

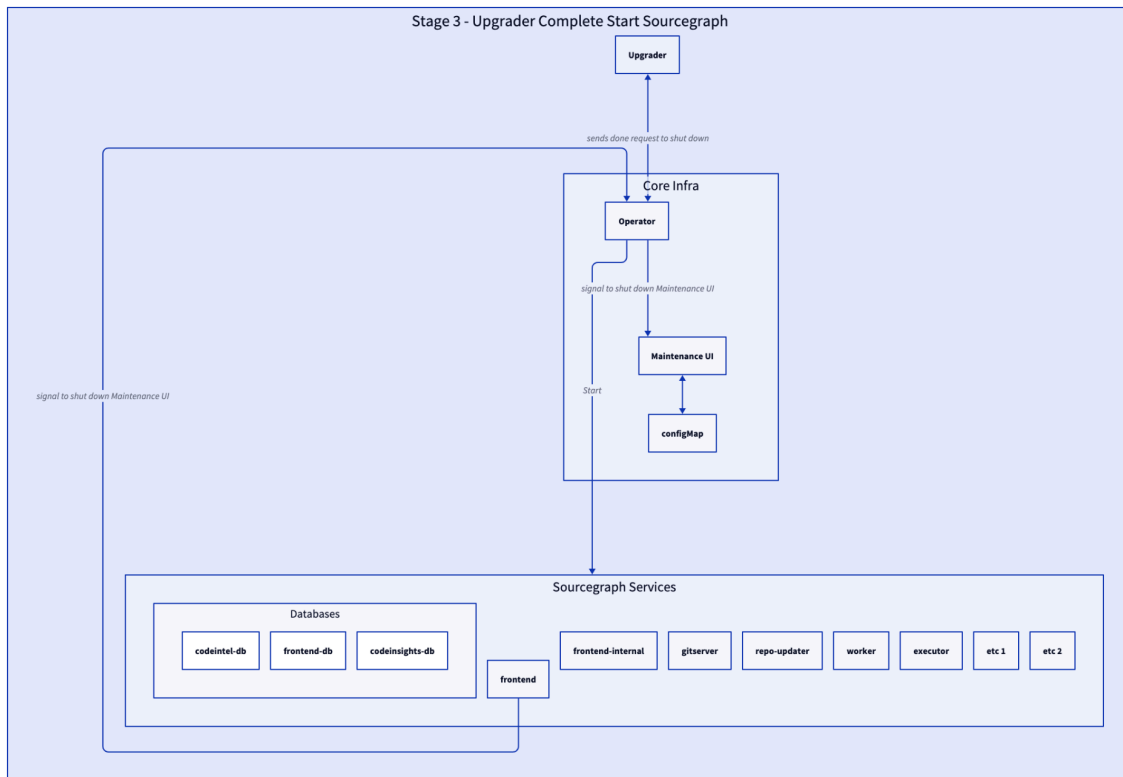
- **Maintenance UI Config Map**

This is the primary source of truth for what is displayed in the Maintenance UI. It will ensure we always have something to show in the Maintenance UI.

With the Appliance initialized and version selected by the user we can look at how the services will proceed with the upgrade.



Here we see a new service **Upgrader**. This is essentially using migrator packages to run necessary operations from the **upgrade** and **up** command depending on what it necessary. This will operate very much in the same way as migrator except that it will also communicate its outputs to the **maintenance UI** to display to the admin the upgrades progress. *Much of the logic necessary here can be reused from **autoupgrade**. [Example](#)*



When the **Upgrader** is complete it signals the **Operator** it's ready to shutdown and **Operator** starts the other sourcegraph services. With this in place we can remove the **migrator initContainer** from the **frontend** deployment manifest.

Accommodating Other Deployment Types

While the main focus of the Release team for the coming 23Q4 and 24Q1 will mostly be on realizing the Appliance Deployment type, we may be able to make some improvements in other deployment types.

Currently the **autoupgrade** type upgrade always attempts to upgrade to the latest deployment. By adding a **target_version** to the **versions** table and providing users the ability to set the target value we can improve the **autoupgrade** experience for admin in the UI. While also preparing for a version selector and button click upgrade which will be available in Appliance.

In order to help navigate any problems in multiversion upgrades an automatic execution of suggested queries from any drift encountered will also be investigated. This will improve upgrades for Appliance and other deployment types. Some [early work](#) has already been started.

Alternatives Considered and Tradeoffs

upgrade-upto

Early in planning an `upgrade-upto` was under consideration to be added to `migrator` this would combine multiversion upgrades and roll `upgrade` and `up` together using the `target_version` definition in the database to determine the correct migration application plan. This could then replace `up` as the default entry command in our main deployments allowing for down up upgrades in all of our deployment types. Ultimately this is similar to the approach already taken with `autoupgrade` however and would still require users to bring down their deployment – in practice many of our admins only run `docker-compose up` during an upgrade so such a command would still need the ability to poison pill the database and would not represent a significant improvement on `autoupgrade`. Essentially being just a refactor.

Switches in frontend

We'll need to introduce more logic in the frontend to make different UI's aware of the deployment type for Appliance. For instance displaying a button for upgrades in Appliance deployments, and suggested actions to users who change the `target_version` in other deployments.

We'll also need some routing logic to handle when the maintenance UI should be served to users/admins rather than the frontend deployment. This will need to happen in the operator logic since the frontend will be down.

Open Questions

- **Interactions with authorization on the maintenance UI:** *We probably don't want regular users to be able to trigger actions in the UI, but they should get a maintenance UI when the site is down.* At the time of writing this RFC we are assuming something similar to the way Souregraph currently works. The first user to register on an instance is assigned admin. For Appliance the first user to access the instance will get the maintenance UI to select the version.

After this we likely won't allow *actions* to be triggered via the maintenance UI. This still needs more consideration though.

- **Env var considerations:** Right now there are a variety of services that take env vars to configure behavior. For example using an [External DB](#) for our databases. Do we want admins to continue configuring these things directly at the manifest, or to have access to

this configuration level via UI? *The appliance model assumes a more limited configuration range than our current deployments.*




Definition of success

How do we know if this proposal was successful? Are there any metrics we need to start tracking?

Migrations to the Appliance?

- Upgrader should have multiple db users, admin for the upgrader, and a regular user that the application will use.
 - Too many credentials
 - Security sweep, issue tickets to teams that have code broken by this change
 - Validate in cluster

RoadMap

1. Upgrades Package
 - a. Create a func `VerifyDatabase(version string, eadOnlyConnString) error`
 - i. Similar to:
<https://sourcegraph.com/github.com/sourcegraph/sourcegraph/-/blob/internal/database/migration/cliutil/validate.go>
 - b. Create a upgrade strategy `func determineUpgradePath`
 - i. This will be used to determine the application of migrator jobs
2. Integrate new upgrade package binaries into Appliance
 - a.  Using the packages above trigger an upgrade via a job issued by the sourcegraph operator – to begin this will just look like triggering a job via a manual change
3. Maintenance UI
 - a.  Consume output from the validation and DetermineUpgradePath UIs to display a minimalist UI when an upgrade path is ready to be applied
4. Appliance UI changes
 - a.  Allow Users to select and set a target version via UI interface in the frontend

Nelson+Warren

- I. Project - NolnitContainers - Cadency suggestion:
 - A. Create validation function
 1. Create a func VerifyDatabase(version string, leadOnlyConnStr string) error
 - a) Similar to:
<https://sourcegraph.com/github.com/sourcegraph/sourcegraph/-/blob/internal/database/migration/til/validate.go>
 2. Package this into a standalone binary (/cmd folder) → print to console given DB **⇒run command line check**
 3. Create /db/status API in the backend that exposes the value above
 - a) Run the function in the API backend initialization
 - b) **⇒call the API and see success/error**
 4. Change the frontend to first of call /db/status API and render an error page if it returns errors **⇒show the error page on problem**
 5. If the VerifyDatabase() fails in the backend, skip loading ALL APIs but the /db/status (either all API available or only /db/status if bad DB)
⇒demonstrate that API is safe and hidden if error
 - B. Create a migration function
 1. Create func (this one CHANGES the database to fix)
 2. Expose as a binary so anyone can migrate/fix if needed **⇒fix/upgrade any database from command line**
 - C. Remove all init containers **⇒no init containers!**
 - D. Integrate with Operator = *upgrader container*
 1. Package the migration and validation functions into an API (in a separate container)
 2. Operator calls when needed **⇒show that we can migrate at will**
 3. Ice on the cake: remove the validate from the API backend and call this service instead

Michael+Warren

No new service just a Operator job

1. Implement POC for Upgrade CRD in the operator:
 - a. it just upgrades the database (trigger by human)
 - b. fancy: maintenance UI and modify ingress on the fly
2. QA, etc
3. Managed upgrade based on Release API

- a. operator will monitor new releases and perform upgrade based on the maintenance window configured by admin
- 4. Remove init container from appliance model
- 5. Ship it