Proposal for splitting a single container image into multiple container images #4661

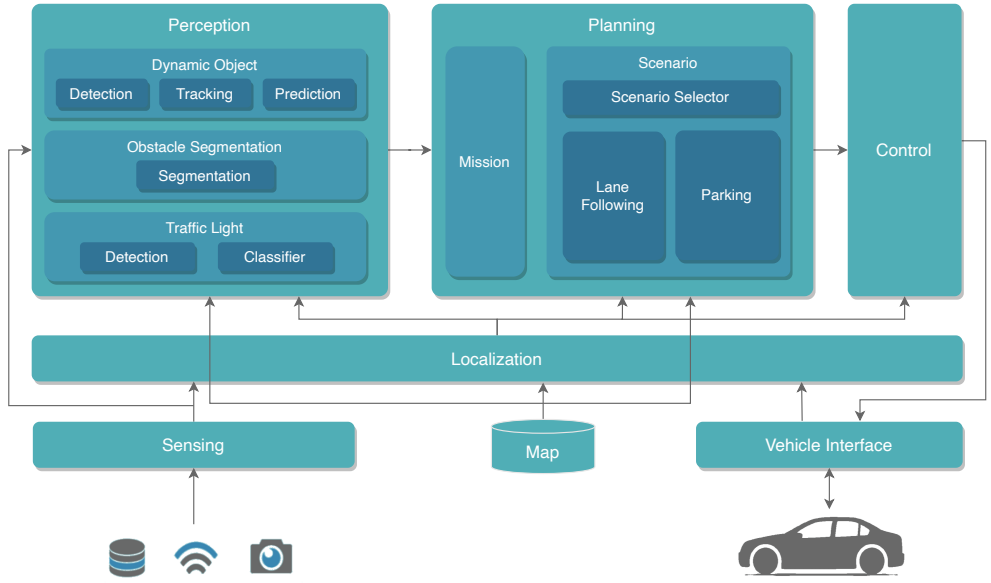
youtalk started this conversation in **Design**

youtalk on Apr 24 Collaborator edited

The only final artifacts in the current Dockerfile are a `devel` image for the development container and a `runtime` image for the execution container. https://github.com/autowarefoundation/autoware/blob/main/docker/autoware-openadk/Dockerfile

As a result, the size of container images can easily become bloated. In addition, modifying a single source code can cause the entire build to be redone, and the caching mechanism does not work effectively.

Therefore, we propose to split container images.
First, create an `autoware-common` image that contains only common message definition packages and common libraries, and run `colcon build` beforehand.
Inherit the `autoware-common` and create multiple container images using multi-stage build for each component of Autoware.
We are planning to configure the units of components according to the Autoware architecture diagram, such as `autoware-perception` for the Perception package, `autoware-planning` for the Planning package, and so on.



Category

Design

Labels

component:openad...

5 participants

Multi-stage builds are expected to reduce the build time of container images and improve cache efficiency. Multiple containerization also makes it much easier to coordinate the computational resources of each Autoware component.
In addition, developers of Autoware components will only need to build the development containers for their own needs.
We believe this change will also greatly improve development and operational efficiency.

This concept is in line with the Open AD Kit, which promotes containerization. We look forward to your suggestions.

↑ 4 👍 5 🚀 5

4 comments · 11 replies

Oldest | Newest | Top

hakuturu583 on Apr 25

I strongly agree with you.

I generally agree with the multi-stage build architecture, but I think there is a problem that modifying packages which was included in the `autoware-common` image or math libraries that may be used inside the various images will increase build time.

To improve the development experience, I think it is important for devel containers to cache apt/pypi packages and compile results using ccache, etc.

↑ 1 👍 1 1 reply

youtalk on Apr 25 Collaborator Author

Yes, thinner `autoware-common` stage becomes better development experience. I'm keeping in mind.

👍 1

doganulus on Apr 25 Collaborator

> Therefore, we propose to split container images.
> First, create an autoware-common image that contains only common message definition packages and common libraries, and run colcon build beforehand.

@youtalk I have a similar view explained in this reply. This direction is excellent and applicable, but package developers and maintainers must address several points before containerization for successful implementation.

1. Containerization will need lists of packages for each container. A mechanism to tag packages by labels may be considered. The OpenADKit group cannot decide which package must go inside or not.
2. Containerization requires stricter dependency management by developers and maintainers. Adding a package dependency to bloat the whole container is too easy, as exemplified in my comment. Currently, 40% of the disk space in the runtime container is devel packages, which is unnecessary. But see that this is caused by `rosdep` install command, starting from the bloated `rclcpp` debian package. See this comment on the ROS forum regarding the container image bloat. The situation has stayed the same since then, as I see. I would appreciate any improvement on that front.
3. Furthermore, developers must always be encouraged and incentivized to **reduce their dependencies to the absolute minimum**, which is necessary for developing safety-critical systems. I would love to see a medium-term proposal from TierIV and everyone else regarding their dependency management strategies.

↑ 1 5 replies

youtalk on Apr 27 Collaborator Author

@doganulus Thank you for sharing the very helpful information.
We will first do the development container splitting. At this point we still have one runtime container. The next step will be to split the runtime container and put it on an image size diet.
I will continue with the multi-containerization gradually and will keep you updated on the progress through pull requests.

doganulus on Apr 27 Collaborator edited

What would be the role of multiple containers in the workflow you think of?

Do you see Autoware developers develop their packages in their component devcontainer? Maybe I am wrong but my impression was that they mostly prefer native development. Yet, if happens, containerized development workflow would be a big achievement.

Runtime images are your end product. Therefore, you need to split runtime before splitting devel containers as developers need released runtime images of other components to develop in isolation. You will save a lot of time if you seriously apply runtime/devel package/container separation without delay. Please don't take the bad example of the ROS packaging practices. That's my number one worry in all that.

youtalk on Apr 28 Collaborator Author

I have already done large scale container development and operations with multiple isolated images on real consumer products as the person in charge.
Also, since Autoware is OSS, it is not practical to change everything out of the blue. Changes need to be made gradually. Separating the runtime container means that the execution method and documentation will all need to be changed. So we will start with the development container, which only affects developers.

doganulus on Apr 29 Collaborator edited

I think runtime images do not have any user base currently. Feel free to propose radical changes. This is a green space.

We want to use Autoware runtime and development container images but we are not able to do it effectively for various reasons, which often comes to the outdated ROS development habits. Containers are a chance to modernize all so we must use this chance well by questioning each and every feature. And discard all unnecessary or replace them modern equivalent. Your experience can be very welcome in all these efforts. I am glad you have joined!

👍 1

youtalk on Apr 29 Collaborator Author

Since multi-stage build is used, it is more efficient to modify them in order from the upper stage. Therefore, we plan to maintain the development container first, followed by the runtime container.
We would like to expand container operations to make it easier for Autoware developers and users.

oguzkaganozt on May 8 Maintainer

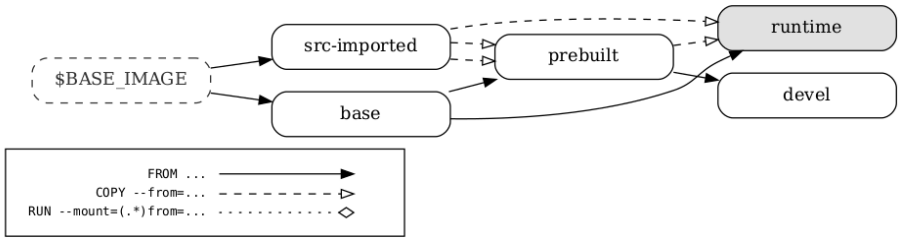
Thank you for this great discussion **@youtalk**, I have created this recent discussion and PR for some initial modular containers (`planning, simulator, visualizer`), please feel free to review it everyone.

↑ 1 0 replies

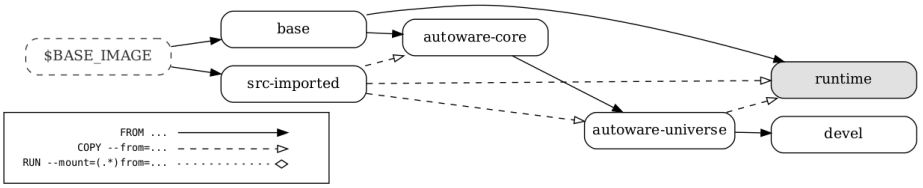
youtalk on Jul 9 Collaborator Author edited

The CI acceleration of the `docker build` has finally settled down.
https://autowarefoundation.github.io/autoware-ci-metrics/
So we will proceed with the multiple container implementation.
I plan to proceed in three steps as follows. The following diagram visualizes the relationship of the multiple stages in `Dockerfile` using dockerfilegraph.

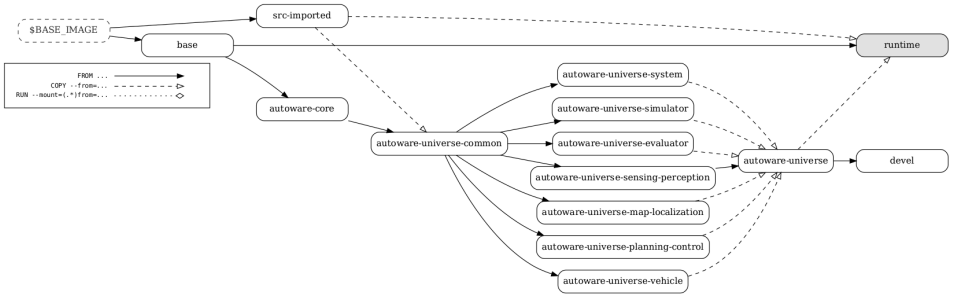
The first one represents the current state.



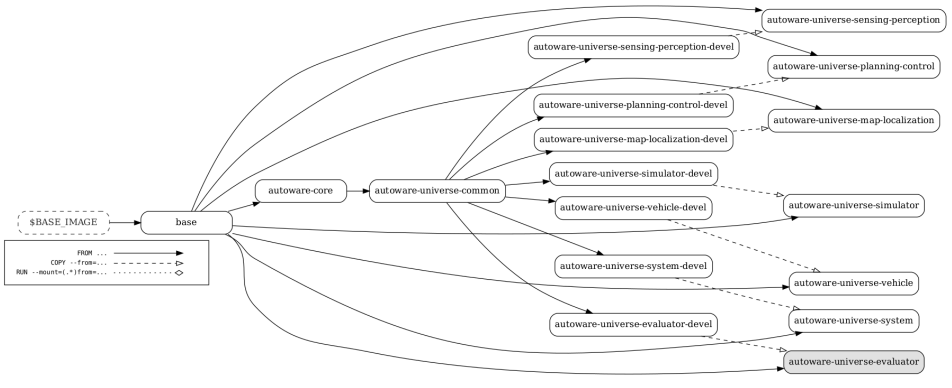
In the first step, the `prebuilt` stage will be split into the `autoware-core` stage and the `autoware-universe` stage. The `autoware-core` includes the packages from the core directory of `autoware.repos`, while the `autoware-universe` includes everything else.



In the second step, the `autoware-universe` stage will be further divided into components. However, in the end, the artifacts will remain the same as in the first step by merging them back into the `autoware-universe` stage. The container build that requires NVIDIA drivers and CUDA drivers will only be the `autoware-universe-sensing-perception` stage. So I hope we don't need to build the other stages twice under the `cuda` and `no-cuda` matrix.



In the final step, the `devel` stage and the `runtime` stage will be removed. Each `autoware-universe-` component will have both a development container and a runtime container. At this time, the method for launching autoware will change from `docker run` to `docker compose up`. Updates to related tools and documentation will also be necessary.



↑ 2 👍 4 5 replies

xmfcx on Jul 9 Maintainer

Containerization benefit analysis and reproducing the results of the paper

I value scientific approach a lot and I appreciate the efforts of TUM researchers on this topic that is paving the way for a more optimized Autoware using containerization approach. But I'd also like to avoid https://en.wikipedia.org/wiki/Replication_crisis and reproduce the results while we still can, because over time the knowledge and readily available setup and know-how to reproduce these results may become less and less available. And I would like to avoid proceeding without reproducing these results.

CPU limitations in the tests

From what I read from the A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation paper, they run the Autoware on a machine with a `AMD EPYC 7313P` which is a `16 x 3.0 GHz (max. 3.7 GHz)` CPU. With https://en.wikipedia.org/wiki/Simultaneous_multithreading (SMT or HyperThreading in Intel CPUs) they would have 32 computing threads. They've disabled it in their tests to avoid fluctuations thus they had 16 threads for Autoware but at least from our experiences with deploying Autoware to real vehicles, we know this is not enough to run Autoware comfortably. Thus I'd hypothesize that containerized approach only improves the latency only on limited hardware by constraining resources provided to the Autoware. On machines with more computing power or if we had a more optimized Autoware *(I'm pointing at you sensing and planning components)* we wouldn't have a performance increase compared to the bare metal approach.

Of course containerization has many other benefits but I'm only talking about the performance aspect here.

Tests also cover the Autoware launch periods

I think this is a big issue to consider. From the paper:

> Each experiment is repeated until 100 valid runs can be evaluated. Each test drive takes approximately two minutes to reach the goal pose.

> **CPU and Memory Utilization.** Fig. 5 shows the distributions of CPU usage measured over all runs, i.e.,, from the execution of the ROS 2 launch file until the vehicle reaches the target position.

The problem here is, Autoware takes like 1 minute to launch due to very un-optimized launch structure. And the test run takes 2 minutes. This means, in every test, they had to wait for very slow launch process. In real life scenarios, this launch procedure is only done once and Autoware keeps running without re-launching for hours. This skews the measurements a lot.

I'd suggest measuring launch stage and runtime stage of the Autoware in different graphs or a long enough run so launch could make up a very small portion of the run.

Right now it is 1 minute launch to 2 minute driving and this is very bad.

For comparison, you can use https://github.com/xmfcx/launch_unifier_ws to simplify your launch file into a single `.launch.xml` file using this and your Autoware will launch within seconds as opposed to minutes. (Same stuff is launched!) Current launch structure is made up of so many python files and is very slow.

Let's reproduce the results on varying computing hardware

I would suggest first reproducing TUM's A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation paper publicly within a single container environment within an Autoware capable system (a computer with at least 24 computing threads). And compare the container vs non-container speed results.

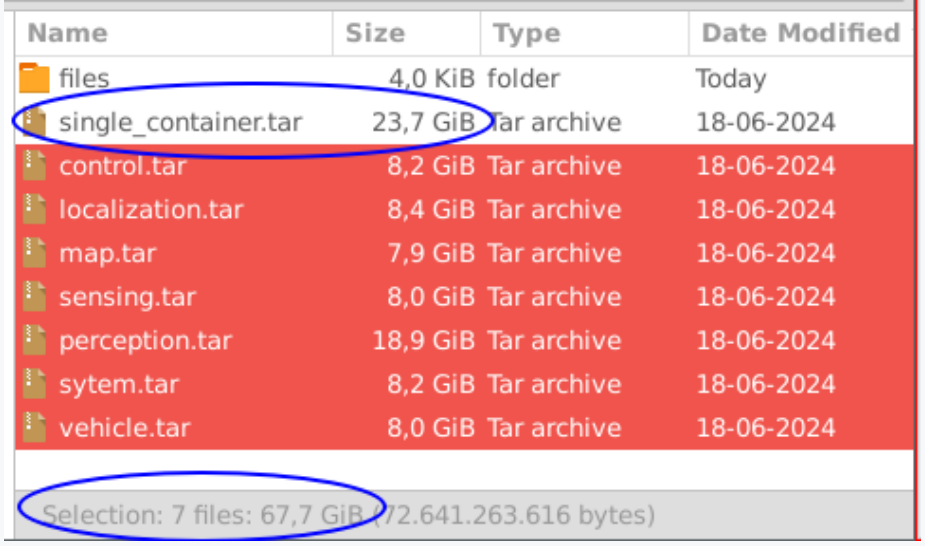
Tobias has provided the images and told me that https://github.com/TUM-AVS/ros2_latency_analysis/tree/dataflow-analysis could be used to analyze the results but I couldn't get more support for it so I've stopped working on it.

Also he stated that they've used a random-removed and traffic-removed version of AWSIM (I've asked for their fork but couldn't get response yet) on a separate machine so that needs to be reproduced as well.

And of course since multi-machine setup is needed, I would suggest setting up a 10G ethernet link and chrony to sync clocks of the machines etc. to eliminate potential network DDS issues, these were not mentioned anywhere.

Potential containerization space issues

Space-wise, multicontainer approach consumes much more space than the single container approach. This is due to duplication of a lot of common libraries that need to be built.
Maybe it can be optimized but I'm not good at containerization area.
Here are the compressed container tar images Tobias provided to me:



I think some of it is unavoidable due to compiled package dependencies, some packages will be recompiled in multiple images.

👍 1

xmfcx on Jul 9 Maintainer edited

Colcon graph of autoware package dependency tree

Here I've customized and shared the process to create a beautiful looking and readable graph of the dependency tree for Autoware:

https://gist.github.com/xmfcx/a39adb809ddc83ef55856eee1ef0a364

You can modify the package selection parts to get to know about a more restricted section of the dependency tree in the following command:

```
colcon graph --dot --packages-skip autoware_cmake autowa
```

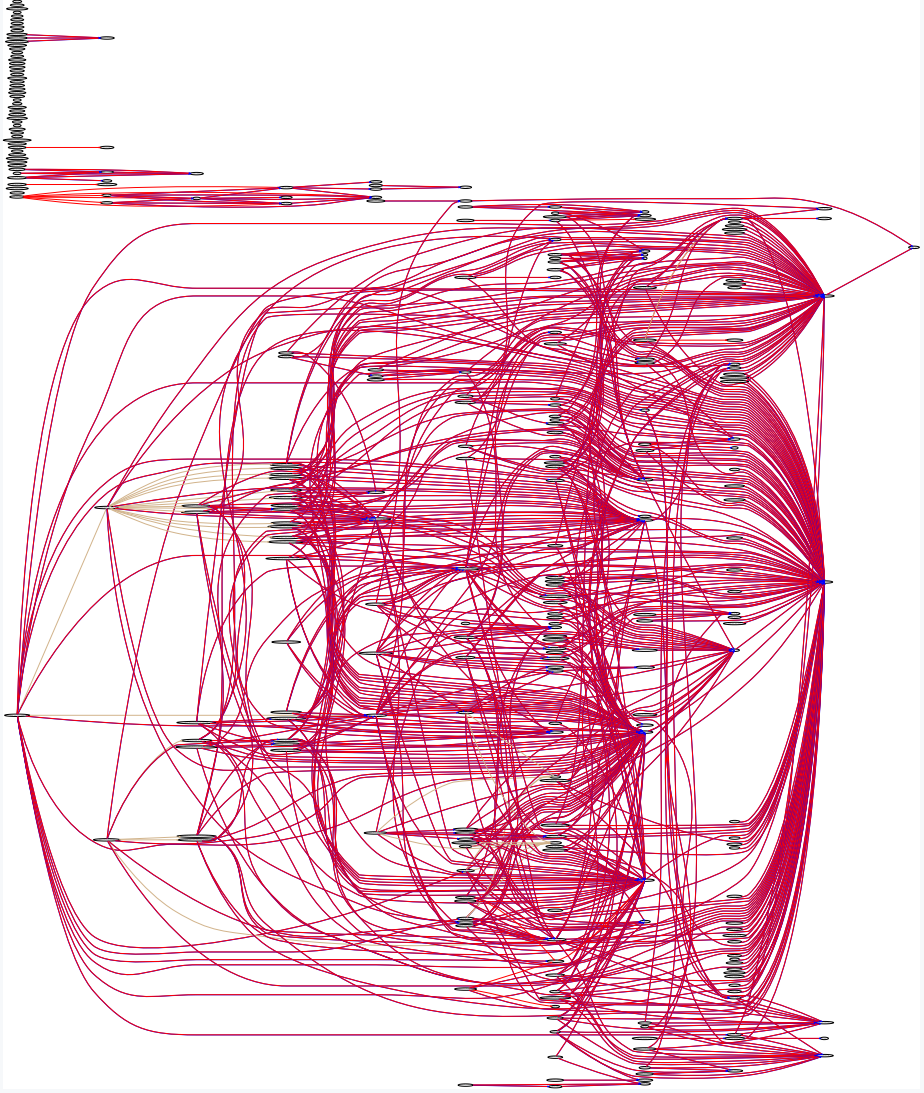
This should be helpful in separating the clusters of packages with package dependencies in mind.

Legend

Colors within the arrows represent:
`#0000ff` =build dependency
`#ff0000` =run dependency
`#d2b48c` =test dependency

Graph

Click to get a detailed look.



👀 1

youtalk on Jul 9 Collaborator Author edited

@xmfcx Thank you for sharing information. It's very insightful for me.

> Right now it is 1 minute launch to 2 minute driving and this is very bad.

How bad is it that it takes a few minutes to start up the entire Autoware system? When operating Autoware in the real world, the startup only happens at the initial run. Do you need to restart Autoware multiple times to continuously run integration tests?
With multi-containerization, for example, you can control the startup sequence, such as starting the `sensing-perception` container first and then starting the `planning-control` container. Therefore, the startup time tends to be longer, but it also allows for more efficient use of the computer and network resources.

> Space-wise, multicontainer approach consumes much more space than the single container approach. This is due to duplication of a lot of common libraries that need to be built.

If multiple images do not share any parent stages, the total image size will be large. However, in the case of Autoware, most of the parent stages are shared, so the total image size is not expected to be several times larger than a single container. Of course, some duplication and redundant layer size increase will inevitably occur, so the total image size could potentially increase by several tens of percent.
However, since each image size will be smaller, the availability for partial image updates or OTA updates will improve.

The multi-containerization procedure I proposed is divided into three steps. In each step, I will proceed with the reorganization, refactoring, and loose coupling of the packages included in `autoware.universe`, while gradually advancing the steps. Therefore, I plan to realize multi-containerization in a way that does not negatively affect performance.

I have experience operating a home autonomous mobile robot with extremely limited computing resources in a similar multi-containerized manner by component. As demonstrated also in the paper, the ability to control computing resources and container startup sequence greatly increased the overall system's availability. I believe the same applies to Autoware.

👍 1

xmfcx on Jul 12 Maintainer edited

I had a very detailed and insightful meeting with Tobias, the lead author of A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation paper. And I'd like to go over my points from my previous post.

About reproducing the paper results

It's not very straightforward to reproduce them, it'd take about 2 full person-months to have all the detailed results from the paper. He also said we might not need to use https://github.com/TUM-AVS/ros2_latency_analysis/tree/dataflow-analysis as is and also maybe use https://github.com/tier4/caret for end to end tracing-latency analysis too. His repo is optimized to re-run tests many times.

He also shared the headless AWSIM setup with me and I was able to build it too.

Personally I don't have time to spare on this task and I'm satisfied with the remaining points. If anyone is interested and ready to take on the task to reproduce the results, Tobias seemed willing to help.

About CPU limitations

Their setup was pretty light-weight with 3 VLP16 lidars (or just one?) and no camera, a typical `AWSIM 1.1.0` setup. And CPU of the machine was on max 30% during the launch and fairly stable 15% during driving. So the CPU was more than enough to run the system even with the SMT was disabled.

> Thus I'd hypothesize that containerized approach only improves the latency only on limited hardware by constraining resources provided to the Autoware.

I was wrong here.

About how launch periods are handled

During the measurements and calculations, the launch times were not included in most of them. So they gave about 1 minute to the launch period and driving took 2 minutes. Launch period generally settled in 40s and it waited for the goal pose for 20s and then the driving & measurements started.

The end to end tracing and latency measurements take a lot of post processing time so it makes sense to keep the driving within reasonable bounds.

Regarding the difference between bare-metal & single-container vs multi-container launch speed difference, I'd predict that if we use the multiple launch files used for multi-container launcher within bare-metal or single-container system, we would see similar results. But we may not know without testing for sure.

From **@youtalk** :

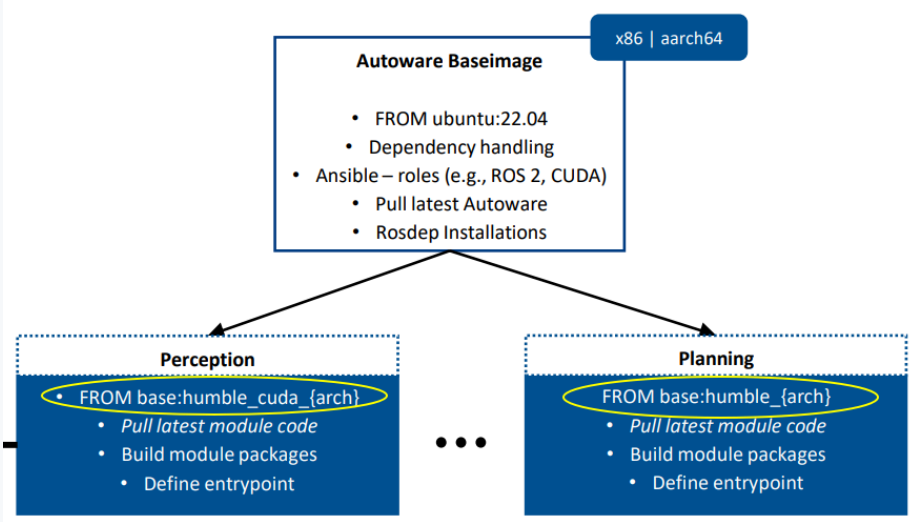
> How bad is it that it takes a few minutes to start up the entire Autoware system? When operating Autoware in the real world, the startup only happens at the initial run. Do you need to restart Autoware multiple times to continuously run integration tests?

My point was regarding the measurements in the tests, not related to how we run the Autoware or how launch duration effects our workflows. Of course we can optimize the launch times to improve the developer experience but that's another discussion.

About the space issue

So the fact that multi container images will occupy more space on the disk is unavoidable. Because the same base image is used by multiple images, the disk space usage is duplicated.

From Fig. 2



But from the update side, if base image keeps unchanged, you only need to download the changed parts, reducing the bandwidth costs, as expected.

Docker multi-stage builds and image sizes

> However, in the case of Autoware, most of the parent stages are shared, so the total image size is not expected to be several times larger than a single container.

@youtalk I have a question:

Let's say we have these images:

- common: 5GB
- perception (`FROM` common): 1GB
- planning (`FROM` common): 1GB

If I pull these 3 images on my machine from scratch, it will download 7GB of data, is that correct?

But if I run `docker images` on my machine, I will see:

- common: 5GB
- perception: 6GB
- planning: 6GB

And on the machine, 17 GB of real disk space will be used, it this correct?

youtalk on Jul 16 Collaborator Author edited

@xmfcx This is not an Autoware-related answer, but a purely Docker-related answer.
Using `docker system df -v`, it can separately display two sizes as following.

- `UNIQUE SIZE`: the actual disk size consumed by the image itself
- `SHARED SIZE`: the shared size that is not added to the real disk size because it is shared with other images

https://docs.docker.com/reference/cli/docker/system/df/#examples

> If I pull these 3 images on my machine from scratch, it will download 7GB of data, is that correct?
> And on the machine, 17 GB of real disk space will be used, it this correct?

Therefore, both the download size and the disk size should be 7GB under the above conditions.

❤️ 2 👀 2