# Intel LLVM/MLIR Collaboration

Intel-wide LLVM/MLIR collaboration, downstream and upstream

**Commented [TE1]:** This document should not just be for MLIR projects but for all of LLVM.

Can we make it less MLIR centric?

**Commented [GR2R1]:** I'm keeping the MLIR in addition to LLVM to make it clear that this is a core part of the effort, but not the only one.

We have not involved the other non-MLIR folks (ex. Phoebe, Aaron) and I would not want them to believe we're forcing them to change how they work based on this document.

**Commented [TE3R1]:** ok I see you added LLVM.

## Introduction

This document aims to help projects find the best way to collaborate across groups and divisions when working on/with the MLIR codebase. The basic assumption is that any code that can go into LLVM upstream, should go upstream. Here we encode best practices for code that is going upstream at different velocities, from different groups, to be shared across Intel for consensus and validation, so we can act as One Intel both downstream and upstream.

There are also projects (dialects, passes, extensions) that should not go upstream[1], but should still live in an Intel repository, so other groups can collaborate on their design.

## How to use this document

The document is split into the following parts:

- **Flowchart**, high-level view of the choices made and the most probable path forward
- **Rationale**, explains decisions to help decision makers pick the best solution for their problem
- **Practical Recommendations**, describing components and how they interact with each other.

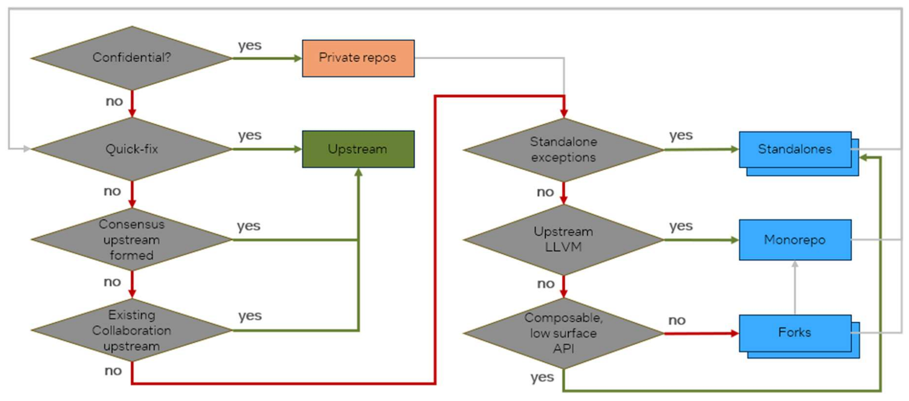
Questions surrounding company-wide and industry-wide collaboration are complex. They can change with time, apply differently within projects, and have outlier costs factored in due to business pressures. The recommendations below are not strict rules, but guidelines on how to navigate that complexity.

It's possible that projects may end in a different mode of collaboration than the ones represented here, and in those cases, the document should be amended to incorporate those modes, so that future decisions can benefit from the added knowledge.

[1] *Some target specific dialects that only connect to Intel only libraries or tools should be kept downstream, as there is no interest upstream for maintaining Intel-specific code that is not used by anyone else.*

## Flowchart

*Ref: MLIR Collaboration Flowchart*



## Legend

### Confidential

Anything that has the potential to disclose confidential, secret, or top-secret information about unreleased products, strategy, or roadmaps. This may require duplication of repositories (ex. public and Intel-restricted), but the confidential repository can be based on the public one, keeping only the confidential parts hidden, so rebase is possible and collaboration can still happen in public.

### Quick fix

Bugs found on upstream LLVM/MLIR that can be fixed directly upstream. These generally don't need consensus or collaboration and can be pulled back from upstream (via trunk rebase) directly once contributed. The work can be one-off, or a continuous stream of fixes, for example bringing up hardware support upstream.

### Consensus formed upstream

Before upstreaming a larger piece of code, it's customary to request for comments (RFC) and discuss with the other members of the community. Once the consensus is reached, patches can be contributed directly without further discussion and it's much simpler and faster to do so upstream. Various Intel groups should collaborate directly upstream on consensus formation, to practice a healthy upstream collaboration ethos.

### Existing upstream collaboration

After consensus is formed multiple times, the team(s) behind the features being upstreamed achieve a stronger, long-term collaboration with other teams upstream. This leads to quicker discussions, less need for new RFCs and sometimes even direct commits without review. When a team is in that state, submitting a PR or committing directly to the upstream repository is always faster and easier, for the items pre-agreed within the team, than staging on an Intel repository.

### Standalone exceptions

Some extensions, dialects and passes require the usage of standalone repositories. The main reasons why this would be so are: different dialects use each other's ops in a complex dependency graph; design uncertainty, experimentation and multiple concurrent options (common in research projects)

**Commented [TE4]:** Please remove the "Uses standalone" decision point. Even if a project uses another project that is standalone if can still go in the monorepo. The real decision point is whether the project iteself can be kept up to date with upstream LLVM, if it cannot that prevents it going into the monorepo.

**Commented [GR5R4]:** The real decision point is if it's meant to be a standalone or not, which has other merits than just upstream LLVM or visibility (as discussed elsewhere). This is what that box represents. Perhaps the description isn't accurate, but we still need a decision point before "upstream LLVM" to decide to use standalone.

**Commented [GR6R4]:** As agreed, changed the description of the node to "standalone exceptions" and described on the legend below the main cases where this would be the case.

By naming it "exceptions" we make it clear that the default choice is still a branch on the monorepo.

**Commented [LH7]:** may change the order with last chapter to make it consistent with flowchart.

**Commented [GR8R7]:** Ack! Done, thanks!

**Commented [GR9R7]:** I'm changing the order in the flowchart to what was in the document because it's more logical to describe reaching consensus before ongoing collaboration.

lead to unreasonable code churn and heated discussions that make it unnecessarily noisy for other projects to cope; dialect staggered upstreaming, where operations go upstream at different times, possibly even into different dialects, need the separation of concerns when upstreaming each new operation isolated from the rest of the downstream dialect; etc.

The key point here is separation of concerns, keeping code that belongs upstream in LLVM directly, making it possible to refine the design of the dialect/pass downstream. Even when not collaborating with other standalones, such projects can more easily distil their design by separating generic code upstream and specific code in the project, until a robust and self-consistent design can be reached and demonstrated.

### Uses upstream LLVM

Projects that use or track the upstream LLVM *trunk*. Intel's monorepo can track the main branch closely and rebase branches on to it with quick turnaround, so project that are tracking LLVM upstream can directly use the monorepo as their baseline. Projects that aim to upstream quickly should also strongly favour the monorepo due to it's fast rebasing and potential for collaboration with other branches, standalones, and forks.

### Composable, low surface API

Projects that add independent new dialects and passes, tools and tests and don't need direct changes to internal source and header files have more flexibility in how they integrate with LLVM upstream and other LLVM based projects.

Projects that neither require a standalone framework nor a special LLVM version can go in the Intel monorepo. However, if they are independent or are meant to be composable, they can still be a standalone repository. Whether it tracks LLVM upstream or some of Intel's monorepo branches, will depend on the project and its collaboration expectations.

**Commented [TE10]:** Sure. Lets add that the monorepo is a good way to distill that design. I do not see the need to keep projects that use other projects that aren't in the monorepo outside of the monorepo.
The monorepo (other than upstream) is the central place for collaboration, so we want to limit as much as possible the proliferation of projects outside of the monorepo.

**Commented [GR11R10]:** The monorepo is not a good way to distill dialect design, as it blurs the contact surface with the rest of the code. This is why standalone is the recommended way to start a new dialect, even if the ultimate goal is to upstream it.

**Commented [TE12R10]:** Why is not a good way ? One can create a branch our of main and work on their dialect. Dialects are pretty separate from other MLIR dialects anyway, they have their directory.

**Commented [GR13R10]:** Renamed to exceptions and added the cases where they would warrant the use of standalone.

**Commented [LH14]:** It is not clear what "our" means here.
Also the sentence seems to describe a scenario that "cannot collaborate ... in the intel monorepo", instead of describing scenario that can.

**Commented [GR15R14]:** Indeed, this whole paragraph is badly written. Let me try to fix it.

**Commented [TE16]:** " However, if they are independent or are meant to be composable, they can still be a standalone repository. "

From the diagram, this decision point happens after one has established the project cannot be in the monorepo (that is it does not use current level of LLVM). We need to make that clear because even if the project has a "low surface API - btw that is a subjective concept) it should still be in the monorepo if it can use current LLVM.

**Commented [GR17R16]:** You keep assuming that the monorepo is the default choice "unless strictly required by other means", which is not true. Standalone is a perfectly valid way to work and collaborate with other standalone and branches. This document is supposed to be a recommendation to the whole company, and must encompass the views of the whole company.

**Commented [TE18R16]:** Yes I am assuming that is case. The status quo is not working, so I believe we have a mandate to make the monorepo a central part of how all teams within Intel collaborate. Again, we can ask for direction and clarification on this because I might be wrong, but I believe that the preference is to use the monorepo as much as feasible.

**Commented [GR19R16]:** In the long run, I agree with you, I just don't think we're there yet. There are practical considerations (CI, Github admin, code review) that we don't have good solutions for in the monorepo and can (and will) be used by groups to discard the whole document if this one small thing doesn't align or interferes with their bottom line.

... [1]

## Rationale

*Ref: MLIR Collaboration Recommendation.pptx*

### Project Directory

We need a list of all our public efforts, to reduce duplication and help formulate the collaboration strategy.

This should become a list of all known projects and their details, at least: name, short description, repository/branch, and admin/code owners (main contact). Additional data can be a longer description, other projects that collaborate with it (internally and externally), existing users/products, etc.

Some downstream work doesn't collaborate at all (ex. connection between internal tool and internal dialect), but they should still be listed in case other groups find it useful or know of a similar effort being done elsewhere / upstream. An extra field may be added to the entry stating its collaboration expectations, to make it clear the kind of scrutiny it may receive.

Not clear where confidential projects will be listed.

### Upstream/Downstream

All that can, should go upstream.

Not all go with same velocity, may need to mature internally (prototype, internal consensus, usability, API).

Downstream work can still collaborate with upstream when visible (ex. target dialects).

Upstreaming Process:

- Prototype/collaborate internally (downstream)
- Propose upstream (RFC)
  - Refused: try a different approach, keep it downstream or drop it
  - Request changes: discuss, perform changes, iterate
  - Accepted: Continue
- Opens collaboration upstream
  - Series of patches that implement the prototype/change proposed
  - Smaller discussions, code review
  - Merge upstream, pull downstream, rebase, test
- After multiple RFCs accepted, less RFCs needed
  - We are now driving design and implementation
  - Can submit PRs directly, no need for RFCs
  - Much faster than holding patch downstream
  - Intel should still review other Intel patches upstream

Prototypes are necessary not only to try out different approaches, but also to have a concrete implementation to help reach consensus upstream. It is, however, a balance. Spending too much time on a prototype that doesn't reach consensus is wasteful.

On the other hand, once a collaboration is open upstream and multiple RFCs have been accepted, it is often more profitable to iterate upstream directly and not need a local prototype. This allows us to collaborate directly with external developers as well as internal ones upstream.

**Commented [TW20]:** Should we list out the expectation of each entry here? Like link to the branch/repo, description, admin, code owners, dependencies, etc.

**Commented [GR21R20]:** Maybe. I don't want to create an expectation this early, so we could just start creating and see what we get, but we could have a minimal set. Do you have suggestions?

**Commented [TW22R20]:** I think a link to the branch/repo, a short description, admin/code owners (i.e., at least one person that we can reach out to when there are problems) should be the minimal set. WDYT?

**Commented [GR23R20]:** Sounds good. Added a few extras as well as optional, just in case.

**Commented [LH24]:** I think we need to clarify whether we should list MLIR projects that build on top of upstream MLIR dialects. Like Plier dialect, optimizations to MHLO and Triton dialects. My opinion is that these projects are not required to be listed, unless these projects also involved enhancement to the upstream MLIR dialects. If we all agree, we should clarify this also.

**Commented [GR25R24]:** If we don't list projects that "are not meant" for upstream, we risk duplicate efforts as groups can decide locally and not advertise widely, and not realise there is already something similar going on elsewhere on Intel or upstream.

I think we should list everything public, but perhaps add a field to the entry stating if collaboration is intended, welcomed or neither.

**Commented [LH26R24]:** I agree to your points. Better to listed all of them to encourage potential collaboration.

**Commented [GR27R24]:** Tried to rewrite with that effect, thanks!

**Commented [LH28]:** Just to clarify the meaning: when the upstream collaboration has consensus about certain idea, the prototype can be skipped?

**Commented [GR29R28]:** Not always. I tried expanding the subject to make it clearer.

Pressure to upstream will likely get higher the more people use it, so should be the interest of the code owners / teams using it to renew their upstreaming efforts to reduce costs.

## Project Types

### Monorepo

**Commented [TW30]:** Can be used in RFC or upstream review to show the whole feature?

**Commented [GR31R30]:** Added, thanks!

For sharing source code changes (see below) that need to keep up to date with trunk.

Allows cross-validation of different work in fewer branches before upstreaming.

Helpful in showing proof-of-concept implementation for upstream RFCs, interacting with external teams, combining multiple features for prototyping, etc.

### Standalone

For sharing with other standalone projects (dialects/passes) when there is more than one dialect. Each in-tree dialect has their own LLVM tree, and none of them can "rebase on each other" without affecting other dialects that do not work with the base, Multiple standalone dialects can rebase on the same upstream LLVM without propagating their changes to unrelated downstream projects.

**Commented [TE32]:** Not quite. A standalone project is one that cannot use upstream LLVM, even if it uses projects that use older versions of LLVM (if it does it can include their headers and libraries).

**Commented [GR33R32]:** I don't understand your comment. Standalone projects can very easily use upstream LLVM. It is not a fork of LLVM, but it uses upstream LLVM directly.

**Commented [GR34R32]:** I tried to clarify what I meant here. The way it was written was indeed confusing.

For working with non-trunk LLVM repositories (customer changes, older commits).

For prototyping ideas before moving to the main tree while keeping clearly separated the project's work from other upstream artifacts that the project generates.

**Commented [TW35]:** This is true for Monorepo and Standalone. Should we duplicate it to Monorepo, or remove it here?

**Commented [GR36R35]:** Just added above.

For keeping downstream dialects that will never be upstreamed (see above).

### Alternative fork

For collaborating with customers/partners on non-trunk LLVM.

For working directly on external forks that do not follow trunk (for business reasons).

### When to use Intel private Github

When the work is confidential and needs to be done in private.

Still uses the types of repositories above, but inside Intel private orgs.

We may have a monorepo inside each org to collaborate on the same visibility level.

## Collaboration Types

### Enhancements to LLVM/MLIR Infrastructure

**Commented [LH37]:** Not sure how this collaboration type can be different than with other types - I mean other types also "direct changes source code". In my minde, we have 5 categories like:
new dialects and associated passes
New passes to existing dialects
Enhancement of existing dialects and passes.
Enhancement of MLIR infrastructure
Enhancement of LLVM

**Commented [GR38R37]:** Right, what I mean here is the last two: enhancements to LLVM/MLIR infrastructure. Renamed and rewritten to make it clearer.

Direct changes to the source code and/or headers of the common LLVM/MLIR infrastructure, including direct changes to existing dialects or passes. These changes can touch internal source and header files which can easily conflict with other changes that use the same APIs.

Internal headers should only affect source in the same directory, external headers can affect anything. If one project modifies an internal API and another uses that API, it may generate conflicts that are not easy to resolve.

LLVM changes can affect any other project in the monorepo, while changes to MLIR or Clang tend to be local (though OpenMP crosses all those boundaries). These changes need to go upstream as soon as possible, as they can also impair the ability to upstream other changes in the Intel monorepo / standalone projects.

### Dialect

New dialects that can go upstream or not.

Downstream dialects can go upstream as a new dialect (or split into multiple dialects) or incorporated into existing dialect(s).

Until there is certainty which upstreaming type, having a standalone project that can interact with all the upstream dialects, create extensions, etc. in a prototype that can be shared with other standalone projects (as libraries and headers) is hugely beneficial.

The end state is to upstream all that is possible and to keep standalone repos for the rest, so they can be used with other standalone in their prototyping.

### Passes

Passes that are applied to upstream or downstream dialects.

**Commented [LH39]:** May separate passes to an individual section.

**Commented [GR40R39]:** Done, thanks!

Upstream passes need to operate on upstream dialects or interfaces. These can also go on the monorepo or directly upstream.

Passes on downstream dialects need to live in the same standalone repository as the dialect and be upstreamed together with the dialect.

### Extensions to Upstream MLIR/LLVM Infrastructure

**Commented [LH41]:** Coud this "upstream logic" be more specific like "upstream MLIR infrastructure"? Besides all your examples, it could include enabling runner, adding tests.

**Commented [GR42R41]:** Added, thanks!

Individual projects that add or extend logic to existing frameworks.

Ex: New types to LLVM IR or MLIR dialects, new data structures, containers, or algorithms to the LLVM API, etc.

These are very intrusive changes that can break any code that is not expecting it.

At an early stage, these would present a huge cost to any work that is based on top of it, especially if the core logic is still in motion (code churn, test failures). Living in the default branch of the Intel monorepo would be very disruptive.

At a late stage, we want to make sure that the other code at Intel works with the new extension. Just before upstreaming we could merge it to the default branch of the Intel monorepo to make sure the rest of the code works well with it (and cleanup the last issues) before upstreaming.

Upstreaming new opt tools and runners, tests and benchmarks are a lot less intrusive and would not affect existing downstream code unless they were implementing the same functionality in a different way.

These problems can be solved by making sure we consolidate our downstream work on the same projects and list them in the projects directory, to make sure teams can help each other instead of duplicating efforts.

### Prototyping

Research, experimentation, pathfinding.

## Practical Recommendations

### Working upstream

Upstream contributions should be monitored by all Intel teams. Requiring notice before upstreaming does not scale and isn't in the best interests of any team.

Intel teams should support other Intel teams whenever possible / reasonable. This does **not** mean approving patches straight away but supporting discussions with strong arguments and evidence.

Disagreeing upstream is not a problem either. It's healthy that we sometimes disagree, it shows that we're serious about technology, that we're not just bumping each other's work regardless.

If there's stronger disagreements, or product roadmap/strategy discussions, these can be taken offline. We can regroup, discuss downstream, and come back up with a unified view upstream. Disagreeing is healthy, arguing against each other is not.

Avoid approving other Intel patches, especially quickly after they've been posted. If your patch is approved by an Intel employee, wait for another approval or at least a few days before committing. We don't want to be accused of fast-tracking incompatible changes.

Make the effort to review other people's (non-Intel) patches, especially in the area you normally contribute. The more we help them, the more they'll help us.

### Working on Intel monorepo

The Intel monorepo is a fork of upstream LLVM with automatic rebase of the main branch. Branches in this fork are used for collaboration between Intel groups and with external collaborators via Pull Requests. These branches can be automatically rebased on the main branch, with a CI job that builds and runs minimal (LIT) testing. Failures on rebase/build/test are sent to the affected people to fix and rebase manually.

The default branch is "intel", where PRs go automatically unless specified on creation. The "main" branch is read-only on our side and constantly rebased from upstream. The rebase of internal branches and the main branch don't need to be in sync. For example, we can rebase the "main" branch every hour from upstream, the "intel" branch every day from our "main", a branch "A" every week from "main" and another branch "B" every month from branch "A".

In addition to basic rebase, build and LIT tests for each branch, there can be extra tests per branch on some groups' own infrastructure.

#### Working on a separate branch

When two or more teams want to work on the same project, they can create a new branch on the monorepo and submit PRs to it from their own forks. The combined work should be a self-contained change and hopefully be composable with other changes.

A PR to upstream LLVM can be made directly from this branch. The changes will be pulled back to the other branches once Intel's main branch is rebased on trunk again.

#### Working on intel's branch

Just like other branches, the intel branch is a place for collaboration. However, its nature is to have multiple PRs from different workstreams to coexist, so a PR cannot be made to LLVM upstream directly.

This branch is for individual commits that can easily be cherry-picked into a local fork and be the base for the upstreaming PR. These tend to be more rapid-fire collaborations than other branches.

### Merging branches and CI

The branch should be rebased on main and built/tested (LIT tests only). It can be rebased/merged with other branches if collaborating further.

## Working on a standalone repository

**Commented [LH43]:** The description looks good to me. Thanks!

Standalone repositories are projects that are not a fork of LLVM, but use its libraries and headers, either as a separate build or as a sub-module. These projects produce additional libraries, headers, tools, and runtimes that can be used by other standalone or non-LLVM projects.

The main reason to create a standalone is to isolate the work from LLVM upstream. Examples are proof of concept / prototypes, self-contained dialect to a downstream library, consensus forming before upstreaming, or a combination of those. It does not preclude or inhibits upstreaming, but it makes it clear what goes up and what stays down, and their dependencies are made explicit.

Standalone projects simplify building and testing and create a clear separation between LLVM code and project code. Ideally, the LLVM dependency comes from upstream (at some recent point), but it can also come from an older revision (or previous release) or even a third-party fork with their own patches on top of LLVM.

The further apart from main trunk the project is, the harder it will be to upstream needed changes, so these situations should be reserved only in case of project/business requirements, for example, interfacing with external tools that have their own requirements.

The LLVM build can come from a separate process (usually projects have a “llvm_version.txt” file and some build instructions), pointing your project’s CMake files to the LLVM build/install directory, or as a Git sub-module, that is built together with the project. Projects can either build LLVM first (like the one above) or build LLVM using the parent project via the LLVM_EXTERNAL_PROJECTS CMake flag.

### How to collaborate with other standalones

Standalones can collaborate if they build on the same LLVM (or close enough). There is no code overlap so no rebase or build issues occur. Each standalone produces a set of headers, libraries and tools that can be used together.

One standalone can import the other as a sub-module, build and take its artifacts, or a third project can be created using two or more standalone projects as libraries. The main concern when collaborating via standalone is which LLVM they all use and how they build it.

For example, if a standalone is meant to be used by others, then it’s beneficial to have a separate build for LLVM, so that it can only be done once for all standalones. If the standalone is a dialect into a third-party tool or library, and it exports a self-contained execution environment, then it can be built inside LLVM as an external project.

### How to upstream changes to LLVM

Given that the code changes are separated between standalone and upstream LLVM, the upstreaming process is slightly more complicated than a standard fork. Standalone projects often have additional dialects and passes, and those cannot be upstreamed.

Upstreaming changes would need to convert the passes / operations to existing ones upstream and then rebased back, converting the internal passes to use the upstream dialects. This is useful to guarantee compartmentalization of upstream versus downstream, existing dialects versus new ones, that is hard to guarantee when working on a fork. The mechanical process is more complicated, but the separation of concerns makes it simpler to know what to upstream and how to rebase back.

### Working on a fork

Business reasons may force a team to work on a completely separate fork. There are a number of reasons why this would be the case:

- A dependency on a third-party LLVM repository (with external changes we don't control).
- If the team needs a separate community (issues, PR, projects, wiki).
- Strong interactions with an external partner that does not want to have to filter the noise we'll be generating on the Intel monorepo.
- The project was not planned to be composable / upstreamable and/or has a definite lifetime.

Regardless of the reason, working on a separate fork is mechanically identical to working on a branch in the monorepo (Git does not distinguish between forks and branches), and collaboration can still occur via *remote* repositories / branches for rebase.

However, working on a separate fork can lead to depreciation of the code, for example if there is no auto-rebase and testing mechanisms. It can also make it harder for the code in such projects to move away from upstream and it will be harder to collaborate with other Intel branches and upstream.

Therefore, while working on a fork is technically similar to working on the monorepo, the latter is encouraged in the interest of keeping the overall costs low to the whole company versus a single group.

Refactoring the code to make it more composable and move to more collaborative development plans is strongly encouraged even if its upstream value isn't clear to the local team, as it may be clear to other teams.

### Cross collaboration

Collaborating across monorepo, forks and standalone is possible and often as simple as if they were all upstream projects. But how it's done will depend on the relationship between the projects, for example, which type of project uses which other type, or if there are more than one user of the same project.

#### How to use a monorepo branch from a standalone repository

This is the most trivial collaboration, and only requires the standalone repository to mark the proposed monorepo branch as its LLVM dependency. Builds can continue in the exact same way as before and no further changes are needed.

Standalone repositories that often collaborate with each other can also create a branch in the monorepo (without any changes) so that they all use the same rebased LLVM code and pay the rebasing costs at the same time, to share the burden. They can even use this branch to upstream a piece of code from one project, but guarantee it works with the others before going upstream.

#### How to use a standalone repository from a monorepo branch

To use a standalone in an existing LLVM fork, use the LLVM_EXTERNAL_PROJECTS CMake flag to build the standalone and the libraries, tools and headers will be available on the build directory. You may need to add some CMake glue if the standalone repository needs something special, but usually it's just a matter of having the repository local and changing the CMake invocation. Check out the IMEX project for more information on how to structure this.

**Commented [LH44]:** Sometimes it is also because the project doesn't make the efforts to make it composable, or making an effort to separate these composable with those non-composable.  Refactoring the code and migrating composable components to standalone repo could be a choice of staged plan.

**Commented [GR45R44]:** Added a few more examples, and a discussion on how to make it more composable at the end, thanks!

**Commented [TW46]:** From the discussion we had in MLIR Collaboration Flowchart.pptx slide 4, you mentioned it is probably not the way one want to collaborate, since it isn't trivial at all.

If it is not the case, then the `Uses standalones` decision box in the flowchart no longer make sense. WDYT?

**Commented [GR47R46]:** This is a good point, actually. My original point wasn't just about building, but maintaining the code.

I have in mind that it is easier to maintain different standalone projects that can share a single LLVM branch than multiple LLVM branches (rebased onto each other) that can share a single standalone. ... [2]

**Commented [TW48R46]:** I can see your point on standalones clearly separates what each project does. ... [3]

**Commented [GR49R46]:** Uses of standalone are not just when you can't use a branch. They're primarily to separate the concerns, either amongst other standalones or fro ... [4]

**Commented [TE50R46]:** My opinion is very simple: if the project can be kept up to date with LLVM then if should be in the monorepo (or upstream directly). Even if the pro ... [5]

**Commented [GR51R46]:** I see no reason to force either way, given that both have their benefits and the costs ... [6]

**Commented [TE52R46]:** I see it differently. Using the monorepo creates more visibility for the project, and should be the preferred way to work unless there severe tech ... [7]

**Commented [GR53R46]:** Visibility is not the only measure. It's not even the most important, since we have the project directory. It makes no sense to require "sev ... [8]

**Commented [TE54R46]:** I am not stating teams have to use the monorepo if there are strong reasons for not doing so (confidential material) or need to use an older versi ... [9]

**Commented [GR55R46]:** As discussed, changed the decision in the flowchart and exposed the exceptions to use the standalone. That is not a fixed list, just some exar ... [10]

**Commented [LH56]:** This is how IMEX supports it- ... [11]

**Commented [GR57R56]:** Added a tip to IMEX on the end of the paragraph. Thanks!

**Commented [LH58]:** "is" is typo? replaced to be "if"?

**Commented [GR59R58]:** Yup, thanks!

Multiple standalone projects can be built as external projects and combined with the existing branch's work, but they all must be buildable against the particular origin branch. Not only the upstream LLVM commit the branch is rebased on, but the changes in the branch itself, especially if it comes from a merge of multiple other branches.

This can create a co-dependency between these projects and needs to be handled carefully.

**Page 3: [1] Commented [GR19R16] Golin, Renato 9/14/2023 5:24:00 PM**

In the long run, I agree with you, I just don't think we're there yet. There are practical considerations (CI, Github admin, code review) that we don't have good solutions for in the monorepo and can (and will) be used by groups to discard the whole document if this one small thing doesn't align or interferes with their bottom line.

I want to avoid strong mandates in the document, so that the onus falls onto the group manager to justify going against the recommendation and their manager to approve the request.

The higher it gets, the stronger the commitment to the document, so I'm hoping that by creating that crumb trail, it'll be harder to make selfish decisions and we'll get what we want.

Politics, I know... :/

**Page 9: [2] Commented [GR47R46] Golin, Renato 9/12/2023 9:56:00 PM**

This is a good point, actually. My original point wasn't just about building, but maintaining the code.

I have in mind that it is easier to maintain different standalone projects that can share a single LLVM branch than multiple LLVM branches (rebased onto each other) that can share a single standalone.

Even though the individual difficulties in building may not be different, segregating the projects in standalones clearly separates what each project does. For example, different dialects or even API changes.

If you create a branch to collaborate with a standalone, you're blurring the lines of the standalone and may make it depend on an LLVM that it cannot escape. And that's a really bad position for the standalone project.

Does it make sense? Maybe @Li, Jian Hui can share his experiences with IMEX?

**Page 9: [3] Commented [TW48R46] Tsang, Whitney 9/12/2023 6:00:00 PM**

I can see your point on standalones clearly separates what each project does.

> I have in mind that it is easier to maintain different standalone projects that can share a single LLVM branch than multiple LLVM branches (rebased onto each other) that can share a single standalone.

That depends on the project dependencies. Assuming project A and B both uses a single standalone, if project A and B don't depend on each other, then their corresponding branches don’t need to be rebased onto each other.

> If you create a branch to collaborate with a standalone, you're blurring the lines of the standalone and may make it depend on an LLVM that it cannot escape.

Why using a standalone in a monorepo branch would make it depends on an LLVM?

I have the mindset of maximizing the usage of monorepo unless impossible, e.g., the project cannot use upstream LLVM. The situation here is given a project that can use upstream LLVM and uses a standalone project, should we suggest them to use monorepo or standalone. Both are possible.

**Page 9: [4] Commented [GR49R46] Golin, Renato 9/13/2023 12:00:00 AM**

Uses of standalone are not just when you can't use a branch. They're primarily to separate the concerns, either amongst other standalones or  from LLVM itself. The example here shows the problem between two standalones, but that is also true if there's just one and you really want to separate the concerns.

One example is when you're developing a dialect and want it to be self-contained. Anything that can be upstreamed into other dialects go out in a branch/fork, upstream, pulled back, and now are completely separate. We use this mode extensively in our TPP experiment and this helps us with upstreaming more than if we were working in a branch, because we have a clear separation.

We have upstreamed to different upstream dialects and will continue to do so, while keeping the separation of concerns between upstream and downstream.

If the dialect now needs to be upstreamed, it should move into a branch (or go directly upstream via a PR from a personal fork) and the rest of the project continues.

With more standalones this is even more important (why I sigled it out), because now you can also separate the concerns across different projects AND upstream. But even for a single project, it can make upstreaming much more efficient than in a branch, where all code is tangled and hard to cherry pick into upstream main.

**Page 9: [5] Commented [TE50R46] Tiotto, Ettore 9/13/2023 10:57:00 AM**

My opinion is very simple: if the project can be kept up to date with LLVM then if should be in the monorepo (or upstream directly). Even if the project uses a standalone project it can still used it by including header files and pulling in libraries.

**Page 9: [6] Commented [GR51R46] Golin, Renato 9/13/2023 4:04:00 PM**

I see no reason to force either way, given that both have their benefits and the costs of collaboration scale equally with the conflict surface, not the number of projects or build type.

Upstream MLIR uses standalone as the default approach to designing new dialects because it allows that separation I mention above. I do not think we should impose a different model on our side.

**Page 9: [7] Commented [TE52R46] Tiotto, Ettore 9/13/2023 11:34:00 AM**

I see it differently. Using the monorepo creates more visibility for the project, and should be the preferred way to work unless there severe technical contraints (can't use recent LLVM) that prevents it.

We can bring this up to the larger group to get feedback on the desired direction at Thursday mtg and move from there.

**Page 9: [8] Commented [GR53R46] Golin, Renato 9/13/2023 5:28:00 PM**

Visibility is not the only measure. It's not even the most important, since we have the project directory. It makes no sense to require "severe technical requirements" to create a project in a perfectly valid way, for perfectly valid reasons, that can still easily collaborate with other teams.

**Page 9: [9] Commented [TE54R46] Tiotto, Ettore 9/13/2023 1:15:00 PM**

I am not stating teams have to use the monorepo if there are strong reasons for not doing so (confidential material) or need to use an older version of LLVM. However I think proliferation of projects outside the monorepo is keen to maintaining the status quo. So if there is a choice my opinion is that teams should use the monorepo.

**Page 9: [10] Commented [GR55R46] Golin, Renato 9/14/2023 5:29:00 PM**

As discussed, changed the decision in the flowchart and exposed the exceptions to use the standalone. That is not a fixed list, just some examples of what kind of exceptions we're talking about. This point should be clear now.

**Page 9: [11] Commented [LH56] Li, Jian Hui 9/11/2023 11:29:00 AM**

This is how IMEX supports it-

Build MLIR and IMEX together from source with IMEX set up as an LLVM external project

```
cmake <config MLIR> -DLLVM_EXTERNAL_PROJECTS="Imex"
  -DLLVM_EXTERNAL_IMEX_SOURCE_DIR=../mlir-extensions
cmake <build MLIR>
```