Continuous Integration for automotive

Principles, challenges and the light at the end of the tunnel

/thoughtworks

Continuous Integration - definition, core principles & practices

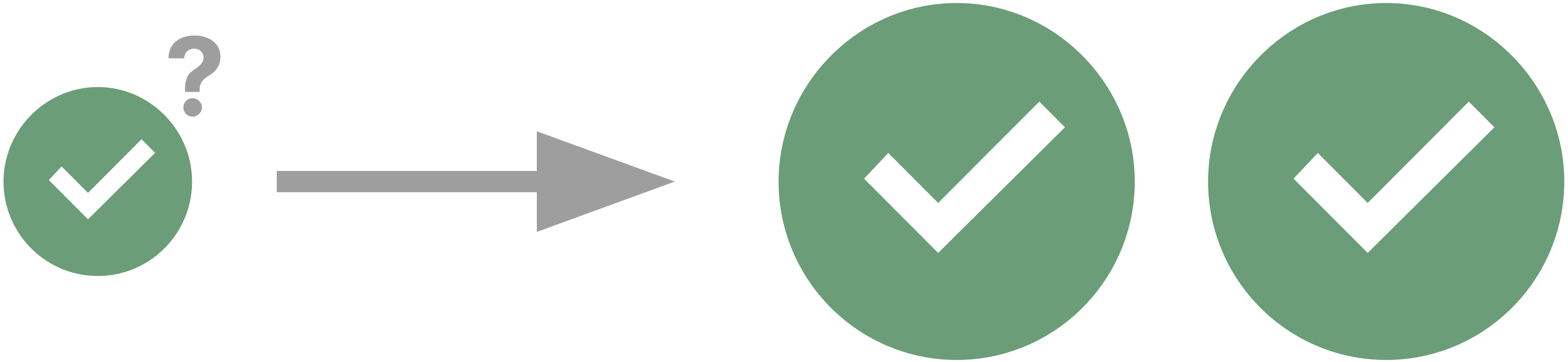
/thoughtworks

WIKIPEDIA

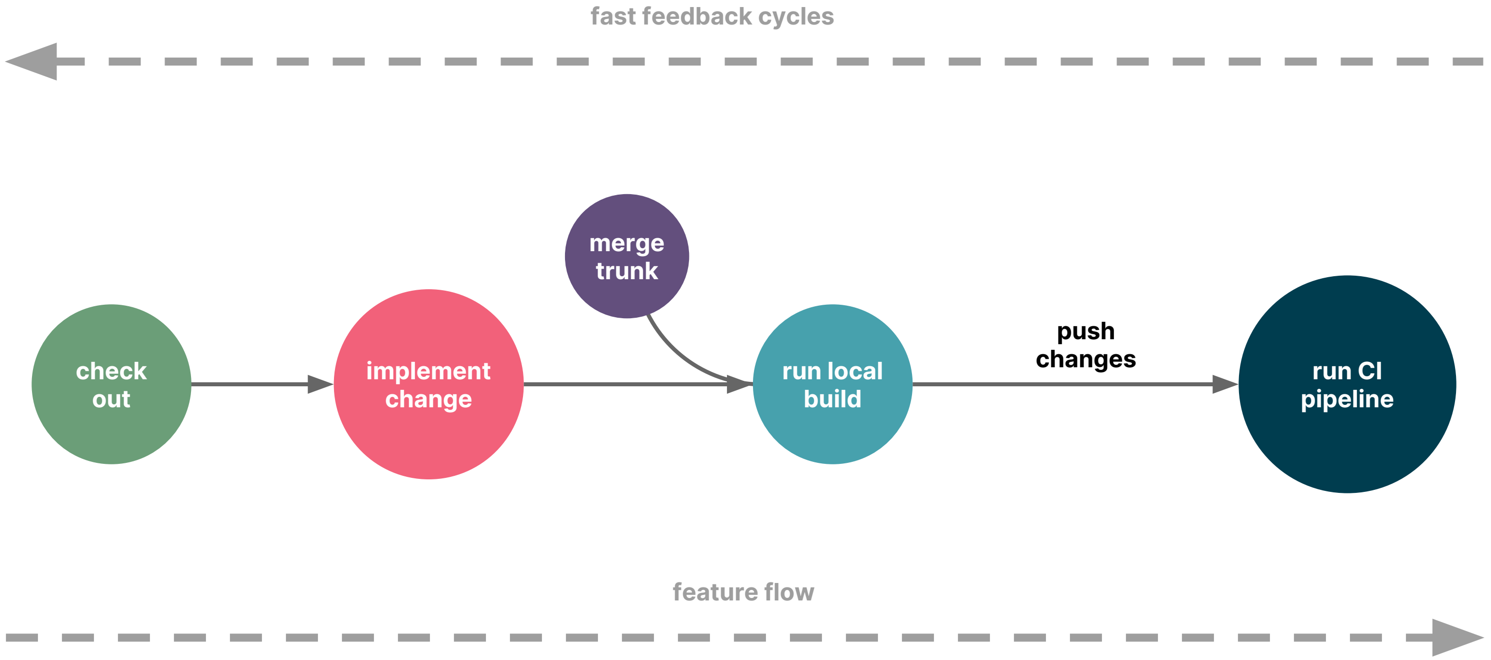
Continuous Integration (CI) is the practice of merging all developers’ working copies to a shared mainline several times a day

Continuous Integration helps to keep your project under control

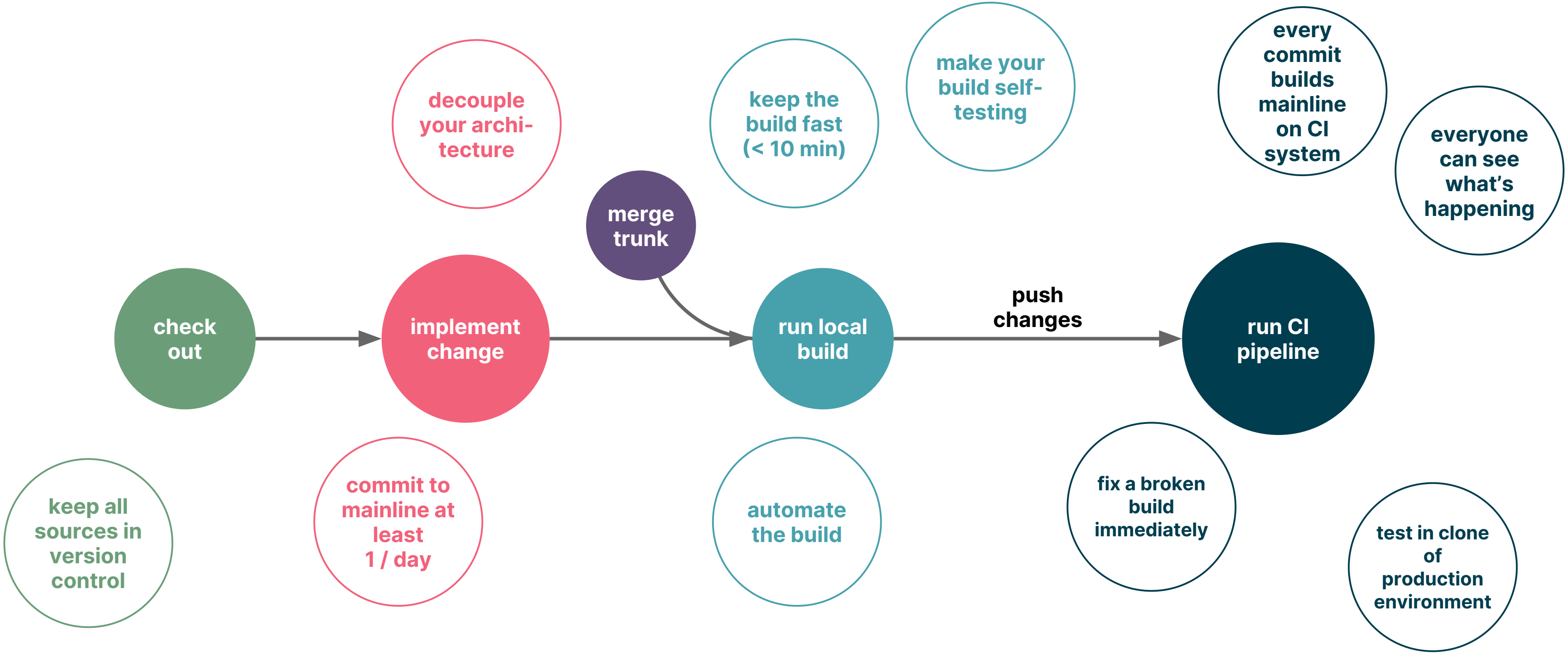
By constantly keeping your code base in a **deployable state** you make sure that implemented **features are "done done"**.



Feature development with Continuous Integration

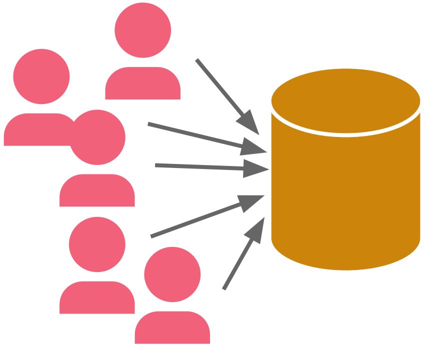


Core principles of Continuous Integration


decouple your archi-tecture
merge trunk
keep the build fast (< 10 min)
make your build self-testing
every commit builds mainline on CI system
everyone can see what's happening
check out
implement change
run local build
push changes
run CI pipeline
keep all sources in version control
commit to mainline at least 1 / day
automate the build
fix a broken build immediately
test in clone of production environment

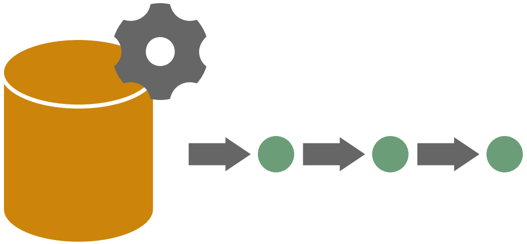
The Continuous Integration Certification Test

1



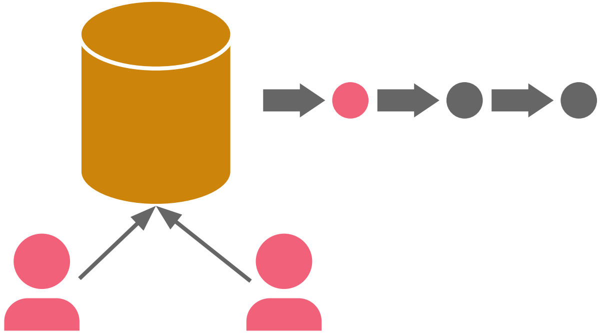
Every developer commits to the shared mainline at least daily

2



Every commit triggers an automated build and test

3



If build or test fails, it’s repaired within ten minutes

see https://martinfowler.com/bliki/ContinuousIntegrationCertification.html

Automotive reality

/thoughtworks

keep all sources in version control

non-versionable artefacts

outdated VCSs (SVN, windchill...)

build artefacts are stored in VCS

decouple your architecture

large batches of changes

application and basis software strongly coupled

cannot be tested in isolation

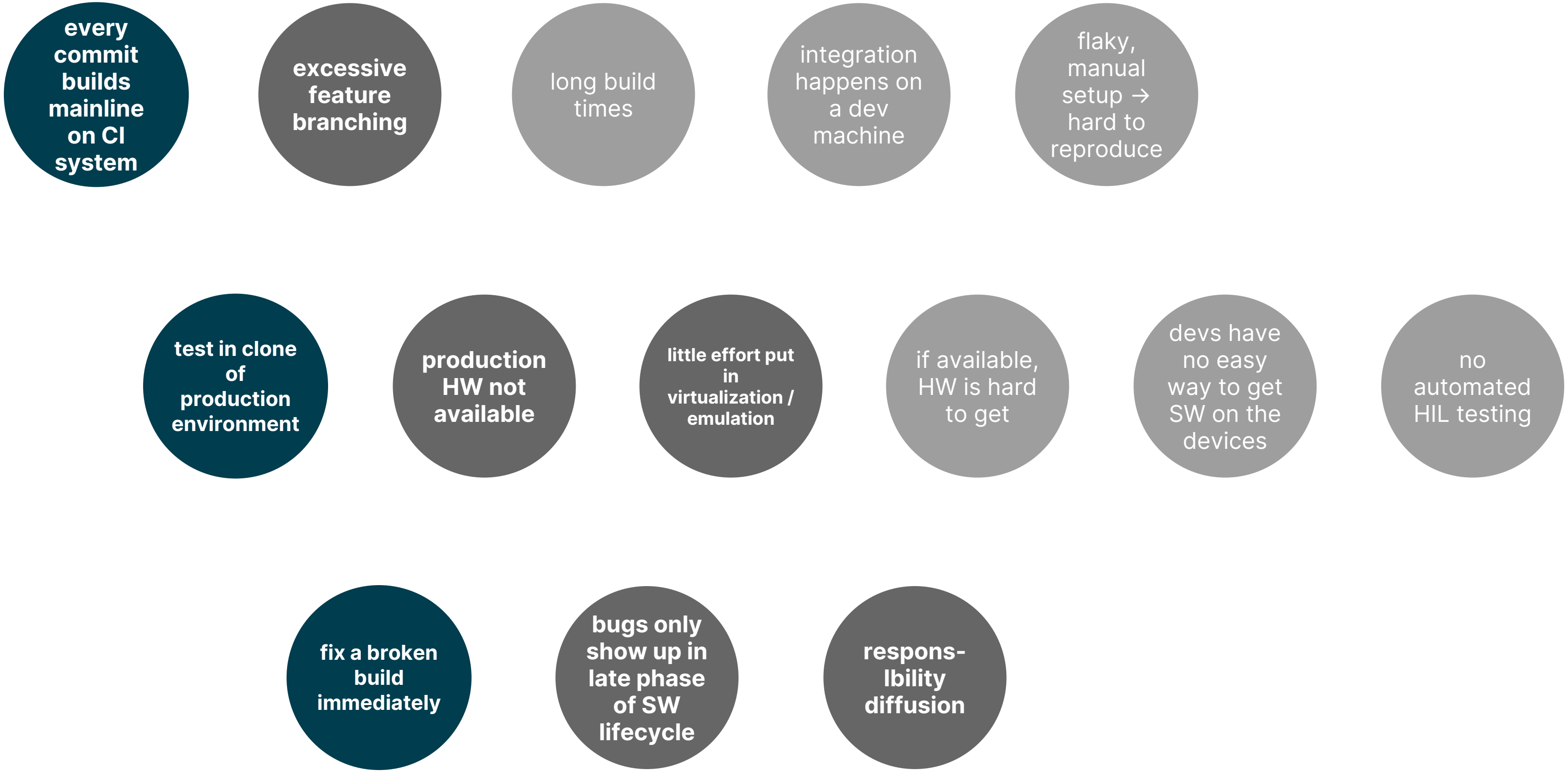
requires lots of upfront planning

commit to mainline at least 1 / day

long living branches

blocking code reviews

run a local build	**tedious build process**	work-station not setup for building SW	build takes too long / too many resources		
automate the build	**manual integration process**	**partially automated build (supplier)**	inadequate tooling for build automation		
keep the build fast (< 10 min)	**complex software stack**	nature of C(++)	(Matlab) models need to be re-generated	strongly coupled components	
make your build self-testing	**separated dev and test teams**	software not optimised for testing	testing as an after-thought	inspection >> build quality in	(too) late testing



every commit builds mainline on CI system

excessive feature branching

long build times

integration happens on a dev machine

flaky, manual setup → hard to reproduce

test in clone of production environment

production HW not available

little effort put in virtualization / emulation

if available, HW is hard to get

devs have no easy way to get SW on the devices

no automated HIL testing

fix a broken build immediately

bugs only show up in late phase of SW lifecycle

respons-Ibility diffusion

And there’s even more...

Beyond the impediments for the core principles that are necessary for Continuous Integration, there are additional - mostly organizational - challenges.

Continuous Integration is no first-class citizen

Comes into the projects as an afterthought. Tools, processes and architecture are not optimized / ready for it.

The “build your own CI” fallacy

Companies often tend to build up their own CI toolchain, which is not competitive with what’s on the market.

The “pipelines team”

Pipelines are not owned and operated by the development teams, but by a “DevOps Team”.

The light at the end of the tunnel

/thoughtworks

Containerisation

Portable dev & build environments

Leveraging container technology can help you to provide portable development environments and establish “dev / pipeline parity”.

Easily spin up development stacks locally	Easy customization of build environments	Re-use development environments in CI	Auto-scaling via container scheduler
Allows for fast onboarding & pain-free local builds and test runs.	Dockerfiles as a standard way of customizing dev & build environments. No more Ansible on VMs.	Allow for the same tools in local development & builds and in your pipeline.	Auto-scaling build nodes becomes easy by leveraging container orchestrators.

Hardware emulation

Beyond virtualization

Allow for building and running your software on the target architecture even on a developer’s machine. Combine containerisation and emulation to unlock another level of CI pipelines.

Faster feedback cycles due to HW independence

No need to connect & flash any HW device. Emulate how your software runs in the target architecture.

Low-cost and ubiquitous availability

Make your developers independent of (non-existing) hardware and provide it at (close to) zero costs.

Reduce the complexity of your CI setup

Run workload on the same architecture, define build environments in code, no exotic infrastructure needed.

Evaluation boards & low cost hardware

Break the vicious cycle of software / hardware dependency

With the standardisation of hardware architectures, we can escape the HW dependency and more easily decouple the hardware / software lifecycle. Eg. Raspberry Pi, nvidia Jetson, Intel nuc...

Allow developers to run their software in a “production-like” environment

Provide early feedback, make hardware available on any desk.

Low-cost HILs

Connect eval boards with your pipelines and get a low cost HIL and faster feedback.

Bring back the “product feeling”

Give developers a sense of how their product feels and acts.

Commercial-of-the-shelf (COTS) CI tools

Don’t invest into systems that don’t differentiate you on the market

Building up your own CI/CD toolchain became unnecessary in most cases. SaaS solutions like **GitHub Actions**, **Azure DevOps** or **GitLab CI** are mature and flexible enough for most use cases.

No more bottleneck to the “internal tooling team”

Rely on existing solutions rather than ticket ping-pong with yet another internal team.

Build on reliable solutions

Leverage existing documentation, community support and Stackoverflow rather than non-existing internal documentation.

Apply proper guard rails

Making your CI workload a fit for established solution might give you good guard rails on “what good looks like”

Everything as code

Apply “everything-as-code” to make your artifacts & tooling compatible with CI

Move away from “none-code” artifacts like databases, de-facto binaries or other non-versionable pieces of information.

Versioning and software tooling

Allow for diffing and thereby rolling back changes. Make changes easy to read by humans. Leverage standard code editors to manipulate any information.

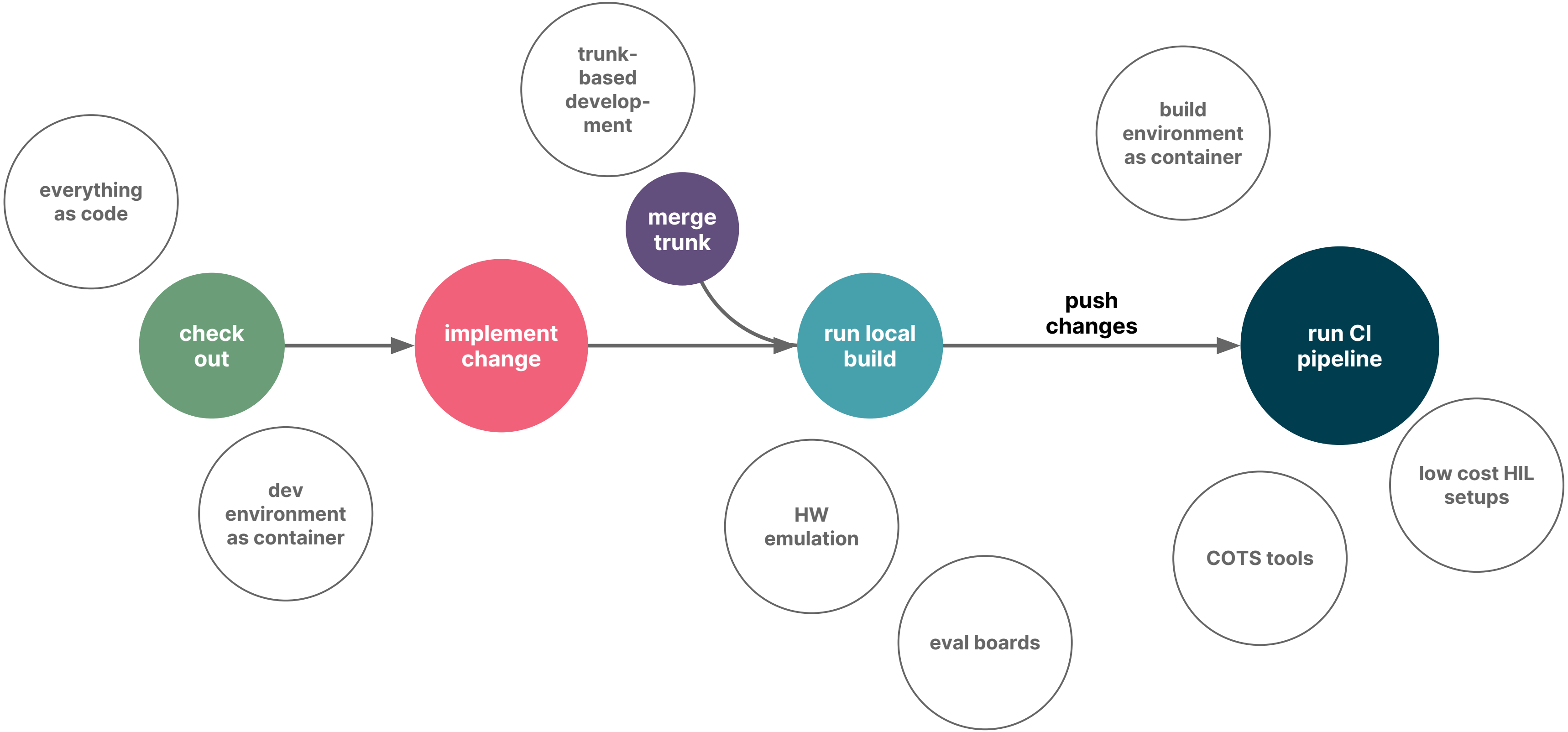
Trigger pipelines via changes

Integrate all changes frequently and incrementally to get fast feedback. Avoid slow release cycles.

End-2-end traceability

Trace changes in your software end-to-end by using the same tracking systems for all artefacts.

CI at the core of your development process



Questions?

/thoughtworks

We look forward to building pipelines with you

Michael Lihs
Infrastructure Consultant
michael.lihs@thoughtworks.com

linkedin.com/in/michael-lihs/

/thoughtworks