2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

ChipDesign DevOps – from sometimes working to almost never broken

Johannes Kösters (koesters@de.ibm.com), Udo Krautz (krautz@de.ibm.com)
IBM Deutschland Research & Development GmbH, Böblingen, Germany
Jeff Brownscheidle (jbrownsc@us.ibm.com), Lou Schmidt (schmidtl@us.ibm.com)
IBM, Austin, United States of America

***Abstract*—Over the past 5–7 years, our global team at IBM has transformed the chip design process from a compartmentalized, domain-specific workflow into a holistic, DevOps-driven methodology. What began as a small-scale automation initiative has now evolved into a standardized CI/CD infrastructure adopted across the entire IBM infrastructure group. The approach has been applied on the chips that are part of the recently announced Telum II®[1][2] Processor, its predecessor Telum®[3] as well as the POWER10®[4][5] processor. These chips are custom processor chips using advanced technology using 500-600 mm^2 of silicon and running at frequencies between 2.4 GHz and 5.5 GHz.**

This transformation spans a multi-hundred-person engineering organization and supports numerous concurrent projects. We discuss the cultural and technical challenges encountered, the strategies used to overcome them, and the quantifiable benefits realized. Our approach emphasizes reproducibility, cross-project consistency, and developer mobility, while integrating industry-standard tools and legacy systems. We conclude with lessons learned and ongoing efforts to further optimize our infrastructure.

***Keywords*—hardware design; workflow; devops; developer mobility; productivity;**

I. Introduction

Traditional chip design workflows are often fragmented, with domain-specific silos and manual processes that hinder collaboration, reproducibility, and efficiency. Recognizing these limitations, we initiated a transformation to adopt DevOps principles in hardware development. This paper presents our experience in evolving from ad hoc automation to a standardized, scalable, and developer-friendly CI/CD infrastructure.

II. Background and Motivation

Our initial environment lacked consistent integration checks, reproducibility, and shared infrastructure. Builds were long, error-prone, and difficult to debug. The higher the level of integration, the more often we experienced a break. With every iteration on chip level, it took us between one and two weeks to recover to a running state on chip level. Inspired by DevOps practices in software engineering, we sought to:

- Automate repetitive tasks.
- Standardize the build and test processes.
- Improve collaboration across domains.
- Eliminate disruptions caused by missing dependency.
- Enable reproducibility and traceability.
- Efficient use of available compute and reuse of results.
- Enable high-quality checks also for external components.
- Revision control internal and external tool versions.
- Abstract infrastructure updates.
- Build a system that can target different acceptance criteria based on the domain or scope.
  - Unit level (team of 2–20 people), i.e., load-store-unit, dispatch unit, cache, fabric, ...
  - Subsystem level (integration of units)
  - Chip or System level (full chip integration)

III. METHODOLOGY AND IMPLEMENTATION

A. Infrastructure Foundation

We began with a small core team (1% of developers) to prototype a Jenkins-based automation system. Over time, this evolved into a company-wide methodology built on:

- Git for version control.
- Jenkins for CI/CD orchestration.
- Make for build automation.
- JFrog Artifactory for artifact storage.
- IBM Spectrum LSF to submit build steps into our compute grid and efficiently manage per job compute resources (CPUs & Memory)

The entire system is operated by a very small team as we enabled the users to efficiently contribute to the overall flow by providing education and keeping the tooling restricted to the above standard components.

B. Feature-Based Development

A key enabler was the shift to feature-based development. This allowed for modular integration, better testing, and clearer ownership of changes. We enabled this by embracing the capabilities of Git, where each feature can get cleaned up by itself (it passes all defined acceptance criteria) on a branch before it gets merged into the design that then flows up to other teams.

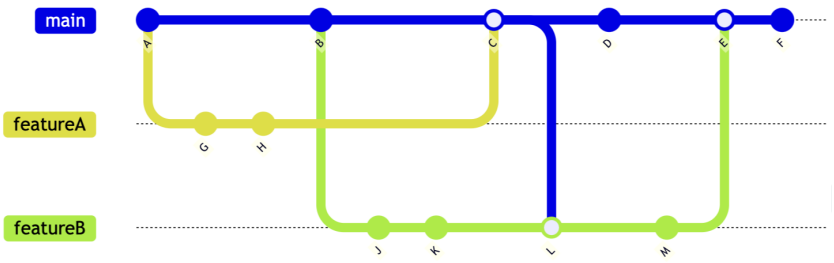

Fig. 1. Feature branches in Git

We built our entire development process around that, supported by management and technical leads. The term 'feature' is meant as a change in design, environment, or even flow. Each feature comes with a well-defined goal and acceptance criteria. The minimum criterion is to pass the CI process that is effective at a given point in time. A feature will begin with a mindset of anticipating who is affected by the change. Depending on the answer, either a few people within a team or multiple teams may have to coordinate efforts. Some examples (ordered by number of affected teams or people):

- A fix for a problem that came up in regression.
- Restructuring of interfaces between units.
- Replacing an entire design component or addition of a new component and its connection on chip level.
- Restructuring the design to address a timing or physical congestion problem that requires RTL changes and therefore affects design, verification, and physical teams across different hierarchies.

C. CI/CD Pipeline Design

Our CI/CD pipelines evolved from minimal checks to comprehensive validation suites, including:

- Design construction.
- Simulation model and environment builds.
- Smoke and acceptance tests.
- Structural checks such as linting and rule checkers.

- Physical design steps and checks.

As a baseline, we defined a structure in which each team (usually in unit granularity) has its own protected branch and CI. Each team is responsible for managing their protected branch, that is, the order of merges and the definition of CI. Our system provides flexibility in these areas. Normally, each team delivers their work products upstream (the next level of integration) through a pull request to the protected branch of that team. The acceptance criteria are extended to include the checks defined by the receiving team.

In addition to running CI, our system provides the capability of running extended checks on snapshots of the respective main branch or by running nightly regressions on a given release of the git repository based on tags. Once teams choose this option, they sign up for monitoring these extended CIs and take steps to fix the problems exposed by those runs. This approach is used mostly to balance runtime of a CI vs. more exhaustive checking, i.e., long-running design rule checks or formal equivalence checks between design transformations. To reduce compute resources, the build system first checks for prior build artifacts using the following steps:

- Get the input hash of all the inputs defined for a given step, extended by the hashes of all the inputs to the steps in the dependency cone.
- Calculate a hash of these hashes and look for an artifact in JFrog for that build step with this hash.
- If an artifact is found, download it from Artifactory.
- If no artifact is found, build the step and upload the result to Artifactory using the hash.

The full system allows for checking that no inputs were missing from the specification by checking that no file gets used from the file system via an audit run. The audit checks are performed as regular pipeline runs with extended tracing and checking enabled. These checks are performed weekly or nightly.

D. Dependency Management

We use make to build our dependencies between our various build steps. To enable hierarchical approaches, we define make targets for each of the design units (smallest team in our system). The overall architecture of our chips defines how units contribute to higher integration levels. Furthermore we defined groups of targets coming from the different skill domains covering logic design (construction and checking), verification (model build, environment construction, smoke and accept) as well as physical design (synthesis, timing etc). Each of the disciplines can then on each hierarchical level define meta targets specifying the checks for CI. These constructs span a build graph. The meta targets serve as anchor points to run CI on the respective hierarchy.

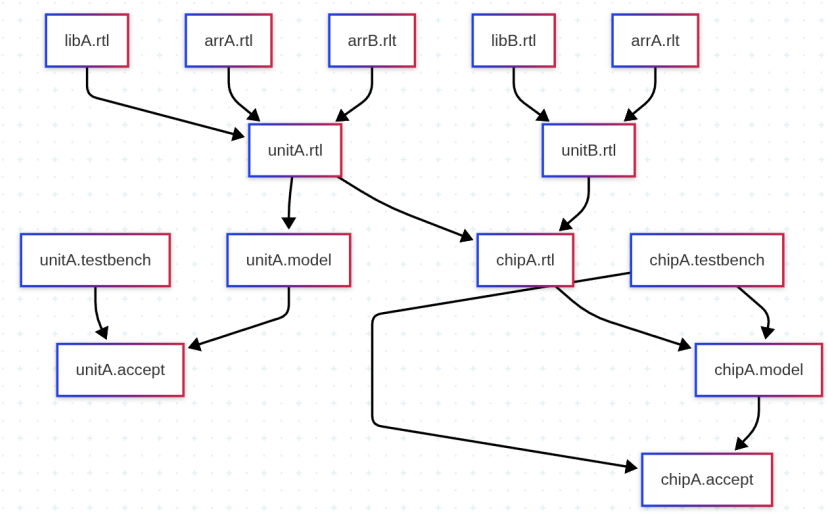

Fig. 2. Example of a dependency graph

IV. Challenges and Solutions

A. Cultural and Organizational Barriers

We encountered common DevOps adoption challenges, many of which are well-documented in software development literature[6]:

Resistance to change and skill gaps: Overcome through early wins, strong leadership support, and continuous education. We put a lot of effort in education and provided class room material and office hours to find help. This were challenging times as we are operating in a global team that spans the time zones of the world from India over Europe to the West Coast of the US.

Our system is also constructed in a way that it is easy to reproduce the results of a CI run locally in your workspace to debug fails and validate fixes while still benefitting from the advantages of artifactory. Over the course of the 5 years venture resistance has turned into a culture that people really do not want to miss what they have now.

Legacy systems: Gradual integration and abstraction layers allowed us to modernize without disrupting ongoing work. Our tools suite is largely using in house tooling provided by an internal EDA team. That allowed us to quickly iterate on issues and changes.

Communication gaps: This is actually something which turned out to be good to uncover. The CI system is helping to expose overlooked dependencies and we see a behavior change in the planning phase. More and more often we are observing that teams are thinking about their dependencies earlier and thus plan for the steps to mitigate breakage from the beginning and not as an afterthought.

The constructs for using artifactory also enforce knowing your inputs better.

B. Hardware-Specific Issues

Applying DevOps to hardware development introduced unique complexities. We were expecting the build times to be much larger than in the software domain and established initial maximum CI execution times. Any build step that is not fitting into that budget we put into pipelines only running nightly or even less often. This technique allowed us to manage the expectations of our users better and enabled us to guide their focus depending on the problem.

The initial budgets we started of with were 5 hours build time on chip level, 2 hours on element level and 15 minutes on unit scope. For most of our environments we were able to achieve these initial goals, in some cases though only by having parallel runs with the different domain CIs on a single GIT pull request.

Conceptually we were seeing the following problems:

Granularity of tasks: Hardware flows often involve large, interdependent steps that are not easily parallelized.

We managed to mitigate some of that by dividing large build steps into smaller pieces in collaboration with our EDA team. Combining this effort with the generic artifactory approach allowed for a better user experience from the beginning as reruns of a given former large step in fail situation often now only have to rerun the failing portion that is affected by a fix.

Reproducibility: Ensured through strict artifact tracking and deterministic build environments. This was a reoccuring problem throughout many of our tools. It manifested in several behaviors and aspects:

- Referencing volatile sources (not write protected design sources, I.e. by referencing designs from arbitrary local paths), which were not closely tracked by git
- We had to change a common practice in our prior tools deployment strategy to use symbolic links to refer to a ‘prod‘ or ‘latest‘ version of a tool. This practice sneaked its way into script and tools developers practice using those tags as version information. As these pointers would change in a global fashion projects saw unexpected breakage ever once in a while.
- Tools themselves not running in a reproducible fashion, i.e. seeing multi thread effects in a synthesis environment. Some of these we have not fixed even by now especially in the physical design domain.

Most often the tools involved are optimization tools that benefit from some randomness in finding a at least locally optimum solution.

This leads to the **Lack of binary pass/fail criteria**: Most of the steps involved in the logic and verification discipline can define a precise pass/fail criteria in a binary form. Especially some of the physical design steps are not as easy to handle in that regard. A synthesis run usually requires checking at multiple dimensions like area, timing, routing congestion, etc. Passing in one dimension may make the other dimension unresolvable. The final signoff on all dimensions is usually only achieved towards the tail end of the design cycle.

To achieve higher quality in these domains we are defining pass/fail as barriers that must be met. Initially the barriers are very relaxed but will be tightened up over the course of the project.

V. Technical Innovations

A. *Unified Build System*

We standardized build commands across CI and developer environments, which enables easy reproduction of CI failures.

The system also simplified onboarding for new developers and allows for a very easy move of engineers from one project to another (developer mobility). It allows every engineer to concentrate on the specifics of the design faster instead of having to worry about how to process it.

Consistent artifact structures and naming conventions, which enables standardized tooling immediately deployable in all of our projects. It simplifies extending the flows and overall tooling. We also paired this with providing means to instantiate external dependencies to further tooling or design sources into the work tree of our git checkout upon make execution. This feature provides a consistent holistic view to all sources leading to the final product instead of having to search through different file system locations or databases.

B. *Artifact Management*

Artifacts are versioned and fetched automatically based on Git hashes. This reduces duplication and increases reuse across teams. It also ensures traceability and reproducibility and enables future extensions like running portions of the build on special compute resources either in the cloud or on special hardware (I.e emulation engines) without having to rebuild or download the full stack.

C. *Integration with Vendor Tools*

We integrated third-party EDA tools (e.g., Cadence, Synopsys). A central system for build is easier to manage from a Licensing and compute requirements perspective than individual users running on their own as we can throttle builds or benefit from artifactory and thus safe licences.

We are facing some new challenges in this domain. It seems to us that vendor tools are often optimized for interactive execution. This manifests for instance in some tools defaulting to a debug shell instead of just returning with an error and a non-zero return code. As much as this is certainly supporting users that do interactive debug it hampers batch execution and it is tedious to find settings for all the involved tools to prevent such behavior. Even returning a non-zero return code upon an error in the execution of the tool is not always given. We found that some tools require parsing a log file for certain markers to figure out whether it passed or failed.

VI. Quantifiable Benefits

While exact metrics vary by project, we observed:

Reduced build breakages: At this point in time we almost never see integration breakage anymore on chip or element level. This is a very important achievement for our higher integration levels as their deliveries are usually on the critical path for every chip design project. Detecting problems early and close to the time a

particular change was made is paying off. In case we detect flaws in the system we continuously improve our checking and extend the reliability of our system.

Faster integration cycles: The availability of a CI/CD system and the culture change in our organization turned out to have a positive effect on our teams ability to handle a couple of concurrent features in parallel. By having clear branch naming conventions in Git, paired with appropriate checking and communication the ability of the team to switch between features, iterate on them fast and still eventually converge without the fear of loosing work results greatly improved.

Improved developer mobility: As the fundamental directory and file structure of each project in our organization is following the same principals and is paired with identical build system constructs and CI requirements and setup our engineers can become productive in a sibling project very fast. They can concentrate on what matters to their work in regards of domain knowledge and micro architecture. They do not have to re-learn how to cope with the design from a build perspective at all

Higher reuse of artifacts and configurations: By enabling users to use artifacts as a standard piece of their day-to-day work, we have reduced the cost of individual rebuilds and also established traceable sources for subsequent steps. Our organization gets less and less dependent on centrally stored data. We have established means to generate caches of such artifacts on top of the simplistic approach of only locally storing the data. In any case we have a much better understanding about how data flows through our workflows by now.

VII. Summary and Outlook

A. *Ongoing Work and Future Directions*

We continue to refine and evolve our system in several areas:

- **CI content and PR execution times**: Balancing speed and coverage.
- **Pipeline modularity and flexibility**: Supporting diverse project needs.
- **Integration of containerization**: For portability and environment consistency.
- **Metrics collection**: To better understand pipeline performance and developer productivity.

B. *Organizational Scale and Adoption*

The DevOps transformation described in this paper is not confined to a single team or pilot project—it has been adopted across the entire IBM infrastructure group responsible for chip design. This group comprises several hundred engineers distributed globally, working on a diverse portfolio of chip design projects. Each project has its own technical nuances, but all now share a unified CI/CD methodology and infrastructure.

This scale of adoption required more than just technical innovation. It demanded:

- **Cross-functional alignment** between RTL, verification, physical design, and software integration teams.
- **Robust governance structures** to ensure consistency and compliance across projects.
- **Scalable infrastructure** capable of handling large compute and memory requirements.
- **Comprehensive training programs** to onboard engineers with varying levels of DevOps familiarity.
- **Standardized tooling and workflows** to enable seamless project transitions and reduce onboarding time.

The result is a cohesive development environment where engineers can move between projects with minimal friction, share artifacts and knowledge more effectively, and contribute to a continuously improving ecosystem. This organizational convergence has not only improved technical outcomes but also fostered a stronger engineering culture centered around collaboration, automation, and quality.

VIII. Acknowledgments

The authors thank all IBM engineers who have contributed their experience and knowledge to improve the environment. We especially thank them for their openness to adopt to new techniques and methods and their continued feedback allowing further improvements.

REFERENCES

[1] *New Telum II Processor and IBM Spyre™ Accelerator: Expanding AI on IBM Z and IBM LinuxONE*. URL: https://www.ibm.com/new/announcements/telum-ii (visited on 08/26/2024).

[2] *Hot Chips 2024: IBM Telum® II processor and IBM Spyre™ Accelerator chip for AI*. URL: https://hc2024.hotchips.org/assets/program/conference/day1/04_HC2024.IBM.CBerry.final.pdf (visited on 08/26/2024).

[3] *Hot Chips 2021: Real-time AI for Enterprise Workloads: the IBM Telum® Processor*. URL: https://hc33.hotchips.org/assets/program/conference/day1/HC2021.C1.3%20IBM%20Cristian%20Jacobi%20Final.pdf (visited on 08/26/2024).

[4] *IBM Reveals Next-Generation IBM POWER10 Processor*. URL: https://newsroom.ibm.com/2020-08-17-IBM-Reveals-Next-Generation-IBM-POWER10-Processor (visited on 08/17/2020).

[5] *HotChips 2021: IBM's POWER10 Processor*. URL: https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_IBM_Starke_POWER10_v33.pdf (visited on 08/17/2020).

[6] Nasreen Azad. "DevOps Challenges and Risk Mitigation Strategies by DevOps Professionals Teams". In: *International Conference on Software Business (ICSOB)*. Vol. 500. Lecture Notes in Business Information Processing. Springer, 2024, pp. 369–385. DOI: 10.1007/978-3-031-53227-6_26.