How VirtusLab Used Bazel and EngFlow to Cut Automotive Build Times by 80% and Optimize CI/CD Costs for a Leading Automotive Consortium

Executive Summary

Modern connected vehicles rely on highly complex software platforms to deliver a seamless user experience. A key component of this architecture is a sophisticated "software bridge" designed to integrate multiple critical subsystems, including audio, entertainment, and cockpit management controls.

Developing this unified platform meant dealing with massive technological fragmentation across development teams. To streamline this process and drastically accelerate the feedback loop for engineers, VirtusLab conducted a comprehensive migration of the platform's build systems to Bazel.

Key Project Results

- **Up to 81% reduction in clean-build time** by distributing fine-grained Bazel actions through remote execution. Representative projects improved from 28.7 to 5.5 minutes and from 9.7 to 4.1 minutes. This success serves as a blueprint to scale compounding savings across the client's portfolio of 20 similar projects.
- **Eliminated redundant repository checkouts** by replacing separate CI jobs for parallel ECU variants with a single Bazel build distributed through remote execution, saving approximately **30 minutes** of aggregate checkout time per single build.
- **3x faster feedback loop** for engineers working on the product.
- **Optimized compiler license utilization** by routing license-dependent compilation actions to dedicated remote worker pools, preventing general-purpose CI workers from remaining idle while waiting for limited toolchain licenses.
- **CI infrastructure future-proofed for the AI boom** (with LLM usage projected to grow PR traffic by 50% every six months, based on our experience).
- **Simplified local development while preserving reproducibility** by replacing a Docker-based SCons workflow with a standardized Bazel interface.

The Challenges

The migration was considerably more complex than a typical move to Bazel. While Bazel is most commonly adopted within a monorepo, the client operated a hybrid repository architecture shaped by several historical reorganizations. Shared platform components, product-specific code, and externally maintained dependencies were distributed across separate repositories, Git submodules, and external Bazel repositories. Additionally multiple logically independent Bazel workspaces located within the same Git repository made build generators like Gazelle challenging to introduce.

These boundaries were driven by factors such as intellectual-property separation, independent development workflows, and integration with large ecosystems including Yocto/BitBake and the Android Open Source Platform. However, the underlying source code remained tightly interconnected and sometimes contained cyclic dependencies across those logical boundaries. As a result, dependency discovery, incremental migration, and build graph standardization were substantially more difficult than in a conventional Bazel monorepo.

While greenfield projects built from scratch under Bazel on the client's side were neat and clean, the earlier-generation ECU software being migrated by VirtusLab still relied on SCons. These builds enforced a global, flat namespace and contained tightly coupled dependencies accumulated over years of development, making them substantially more difficult to migrate than greenfield projects. Attempts to accelerate this work using LLMs without sufficient project-specific context and architectural oversight produced inconsistent build definitions and quickly became difficult to control.

The primary bottlenecks at the starting point were:

Technological Fragmentation → The client's development ecosystem used several build systems, including SCons, Yocto/BitBake, Gradle, CMake, Cargo, Go tooling, and Soong for Android build system. Bazel had already been adopted for newer projects and was selected as the strategic build system for consolidating this fragmented stack. VirtusLab's primary focus was migrating the most complex ECU projects from SCons without disrupting ongoing development, which required a phased, risk-mitigated roadmap.

Insufficient Test Coverage for Migration Validation → Older driver code for earlier vehicle models had very low unit test coverage. Many existing "unit" tests were effectively integration tests that required substantial portions of the production codebase to build, while others relied on mocks, stubs, or generated code that did not accurately represent production behavior.

Consequently, the only reliable way to verify a successful migration was to compare the newly compiled outputs with the legacy binaries, bit for bit. The alternative was end-to-end testing on a limited number of dedicated vehicle-simulation rigs, which required a complete ECU build and could therefore be performed only after the migration was finished.

No Reliable Module-Level Validation	→	A tight web of interdependencies made it impossible to validate migrations incrementally, one module at a time. Individual modules could compile and pass their available tests while still failing during final linking because of missing object files, conflicting symbols, incorrect dependencies, or incorrectly selected sources. Consequently, migrations frequently became large, atomic changes - a single module migration could require adding or modifying up to 700 Bazel BUILD files and incorporating approximately 2,000 compilation units.
CI/CD Bottlenecks	→	Every Continuous Integration (CI) run consumed massive resources. A single build on traditional AWS infrastructure required 5 hours of actual machine execution time (12 parallel machines running for 20-30 minutes each), driving up cloud costs and delaying development teams across the US and Europe.

The Solution:

Standardizing on Bazel and EngFlow, using structured, human-in-the-loop approach to AI-based migration

While the client had already chosen Bazel as their target build system, the initial implementation suffered from heavy "retrofitting" - forcing legacy build patterns into Bazel instead of adopting its native paradigms. This created massive technical debt.

A key example was the handling of preprocessor definitions that controlled constants, included conditional headers, or selection of logic for variants of the ECU.. The previous build system generated these flags from a configuration file and applied them globally. The initial Bazel implementation preserved this mechanism by passing the generated file through each target's copts. However, copts and local_defines apply only to the target that declares them, while defines propagate to dependent targets. Every affected target therefore had to reference the configuration file explicitly; one missing reference could silently change compilation behavior. Correcting this globally would also have broken existing builds whose source code had come to depend on the non-propagating behavior, requiring careful review by domain experts.

To resolve these deep architectural issues and scale the build system, VirtusLab introduced a structured migration framework:

Step 1:

Structuring the Build System Migration Into a Phased, Risk-Mitigated Roadmap

To structure the build system migration, we developed the plan including migration of smaller, actively developed projects built with Make/CMake, Go, and Rust/Cargo. While these modules were simpler and relatively well-tested, they suffered from a massive feedback bottleneck - often requiring weeks of waiting for manual verification.

Step 2:
Implementing a Two-Stage Binary Validation Pipeline

To prove the migration was correct without relying on slow, physical vehicle testing, we designed a rigorous, automated verification process:

→ **Stage 1:**
Object File Alignment

We systematically compared compiled object files for every single source file, matching significant binary code sections. Any discrepancy immediately pointed us to mismatched dependencies, source files, or unpropagated compiler flags.

→ **Stage 2:**
Forcing a Deterministic Linking Order

While source files can compile in parallel, the final binary layout depends strictly on the sequence in which the resulting object files are linked. We engineered a custom configuration that forced Bazel and the legacy build tool to align their linking sequences exactly. This achieved absolute bit-parity between the old and new binaries.

Temporary Bug Preservation:

To achieve this absolute bit-parity, the team had to deliberately preserve certain historical, buggy behaviors of the legacy code. Without reproducing these exact quirks, validating the migration through binary comparison would be impossible. These temporary workarounds are slated for cleanup as soon as the final legacy project relying on this module is fully migrated.

Step 3:
Integrating EngFlow for Distributed Execution and Caching

EngFlow is a remote execution and build caching platform that distributes Bazel actions across remote worker pools and reuses previously computed results across CI and developer builds. The client had already adopted EngFlow as part of its Bazel infrastructure, but the migrated ECU projects still needed to be adapted to take full advantage of it.

VirtusLab resolved issues specific to remote execution, improved the portability and granularity of build actions, and optimized how workloads were distributed across the available infrastructure. Fine-grained action parallelism significantly reduced clean-build times, while remote caching accelerated incremental builds and eliminated repeated work across developer and CI environments.

Step 4:
Deploying a Human-in-the-Loop AI Migration Framework

Earlier attempts to accelerate the migration with LLMs produced changes spanning hundreds of build files, making them difficult to review thoroughly and allowing architectural inconsistencies to accumulate. Our engineers designed the end-to-end AI-assisted migration workflow, defining how project context was provided, which tasks could be delegated, how changes were generated, and how their correctness was reviewed and validated. AI was then used for narrow, repeatable tasks based on project-specific context and pre-validated templates, while experienced build-system specialists handled complex dependency structures, SCons-specific behavior, and cross-module architectural decisions.

The Results:

Build system suitable for LLM-native development

The combination of Bazel and EngFlow delivered immediate financial and operational benefits, drastically improving the Key Performance Indicators (KPIs) of the client's infrastructure..

Metric	Legacy Build System	New System (Bazel + EngFlow)	Gain
Full Build Time (24 variants)	20 - 30 minutes	**<10 minutes**	**70-80% Reduction**
CI Machine Time (per build)	~5.0 hours	**<10 minutes on one orchestration runner**	**96% reduction in client-side runner**
Developer Feedback Loop	Slow (machine queues); ~1 min startup overhead (SCons/Yocto analysis) + full rebuilds due to no local cache.	**3x faster** through local incremental builds and cache reuse	**Shorter iteration cycles and productivity boost**
License Management	CI workers remained **idle while waiting** for limited compiler licenses	License-dependent actions routed to **dedicated remote worker pools**	**Better license utilization** and less idle CI capacity

Based on our experience, we see that moving to AI-based software development can increase the number of PRs even by 50% every 6 months, putting a huge strain on CI/CD infrastructure. That’s why it’s important to have a solid and scalable foundation - what was once nice to have, now is crucial in efficient and modern software development.

Compared with the previous SCons setup, where centralized configuration and a flat include namespace made dependencies largely implicit, the migrated Bazel build requires each **cc_library** to declare its dependencies explicitly. Colocated BUILD files give engineers and AI tools a locally understandable build graph without requiring them to reconstruct the entire global build context.

Meanwhile, EngFlow’s advanced caching and work balancing ensures that even with a massive surge in CI requests, infrastructure costs grow slower than linearly, as repetitive operations are served from the cache in fractions of a second.

Conclusion

The success of this migration for a global automotive consortium demonstrates that modern development productivity in embedded software does not come from simply adding more cloud resources or applying AI tools without sufficient context and engineering oversight. It depends on deep build-system expertise, architectural standardization, and the disciplined use of AI-assisted workflows within clear, project-specific guardrails, supported by technologies such as Bazel and EngFlow.

With VirtusLab's support, the client gained a faster and more scalable build platform, reduced avoidable CI overhead, and improved the development feedback loop. More importantly, the new architecture provides a stronger foundation for handling the expected growth in development activity and CI demand associated with AI-assisted software engineering.

About VirtusLab

VirtusLab helps engineering-driven organizations improve developer experience and modernize complex build systems at scale. Our Bazel experts design, migrate, stabilize, and optimize build environments across monorepo and multi-repository architectures, non-trivial CI/CD pipelines, remote execution, caching, and developer tooling.

We combine deep build-system expertise with Developer Experience strategy and carefully designed AI-assisted workflows. Our goal is to make software delivery faster, more predictable, and easier to scale - reducing build and feedback times, improving reproducibility, and creating engineering foundations capable of supporting both human and AI-assisted development.

Work with VirtusLab to turn your build system from a development bottleneck into a foundation for engineering productivity.