車両ソフトウェア開発でのGitHub Enterpriseの活用事例紹介*

Case Study: GitHub Enterprise for Vehicle Software Development

望月 洋二
Youji MOCHIZUKI

Software development for in-vehicle systems has seen a rapid increase in the line of source code, caused by advanced safety systems and autonomous driving. With the growth of developers and development locations, the use of a centralized repository like Subversion for source code management has become limited due to its distributed locations. To overcome this limitation, the development environment has been enhanced using GitHub Enterprise Cloud, provided by GitHub. This distributed code management system improves code quality and efficiency, as well as facilitates collaboration among developers and development locations.
In addition, GitHub Enterprise Cloud has Continuous Integration and Continuous Deployment tools called GitHub Actions. We apply these tools to increase software development environments.

Key words :
Git, GitHub, CI/CD, DevOps

1. はじめに

自動運転・先進安全システム（Automated Driving/Advanced Safety Assistant System：以下 AD/ADAS と略す）は，自動車業界に革新的な変化をもたらしている．AD/ADAS の機能が高まるにつれてソフトウェアの役割が増大し，複雑性や品質要求も高まる．ソフトウェアは，ソナー，カメラ，ミリ波レーダー，ライダーなどの様々な入力データを処理し，適切な制御信号を出力する必要がある．また，周囲の環境や交通状況や法規の変更に応じて，柔軟に対応できるようにソフトウェアが更新可能であることも求められる．

Fig. 1 に示すように，このような高度な機能を実現するためにソフトウェアの開発規模は増大し，ソフトウェア・コードの行数は 2017 年には 1 億行を超える行数となっている．より高度な AD/ADAS に対応するため，今後も他の組み込み機器と比較しても加速度的にソフトウェアのコード行数が増加していくことが見込まれる [1)]．

ソフトウェア開発規模の急増に対応するため，それに合わせて開発者の人数や開発拠点を急速に増やす必要があった．Fig. 2 に示すように，従来はデンソー本社の 1 フロアで収まっていた開発チームも，北海道から九州まで，場合によっては海外のソフトウェア会社とも協力する必要がでてきていた．開発者の拡大，開発拠点の拡大にあたり，いくつかの課題が見えてきた．

まず第一に，デンソー社内のサーバーに構築された中央集権型のソースコード管理システムに接続するためのインフラの構築に時間がかかること．ソースコードのバージョン管理システムは，いまやソフトウェア開発において欠かせないものとなっているが，サーバーに接続するためには専用線の導入やセキュリティの監査などが必要となり，半年以上の期間がかかっていた．

次に自己流でソースコード管理システムを使う人が出てきてしまい，それを防ぐ手段が少ないこと．ルールで決めていても，それをシステム的に防ぐ手段がなく，開発者の善意に依存する運用となっていた．ソースコード管理システムはリポジトリという単位で管理されているが，例えば，リポジトリへの登録（コミット）を共有ファイルサーバへのコピーと同感覚で使われても，それを防ぐ手段がない．

最後にソフトウェアをビルドする環境の再現が難しいこと．従来であれば特定のビルド環境，特定の人がビルドを実施すればよかったが，拠点が分散すると各拠点で同一のビルド環境を構築する必要がある．また，リアルタイム OS やコンパイラ等のソフトウェア開発キットも随時更新されていくため，過去にリリースしたバージョンをメンテナンスするためには，ビルド環境も適切なバージョンにロールバックしてビルドしなければならない．

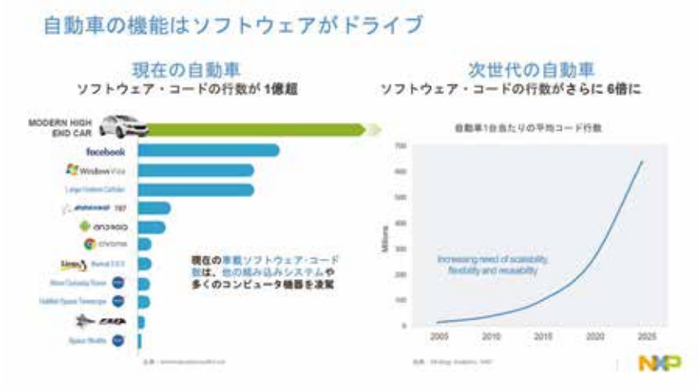

Fig. 1　Increase software code volume on a car

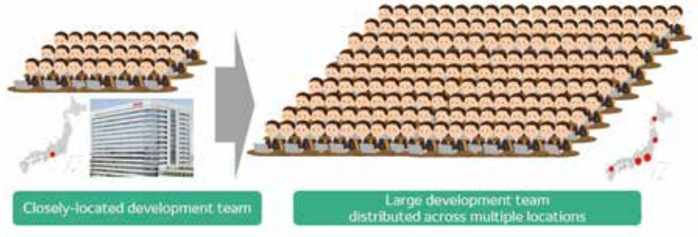

Fig. 2　Developers and centers from before to current

2. インフラ構築の課題

開発拠点の増減に柔軟に対応することは，ソフトウェアの開発規模の急増に対応するためにも重要である．開発拠点の増減に柔軟に対応するためには，新たな拠点拡大時のインフラの整備期間を開発着手の準備期間と同じレベルまで短縮する必要がある．当時は，セキュリティの監査に 1 か月弱，その後機材の調達と専用線の回線工事に 5 か月以上かかっていた．さらには協力会社のセキュリティポリシーにより追加の対応が必要になることもあり，追加で数か月の期間が必要になることも少なくない．専用線が開通するまでの期間は電子メールやファイル共有でソースコードをやりとりする必要があり，デンソーの社内ネットワークにアクセスできるデータの受け渡し専用の人員を追加で確保する必要があった．

＊ Design Solution Forum 事務局の了解を得て，Design Solution Forum2021 発表資料を加筆・修正して転載

特集

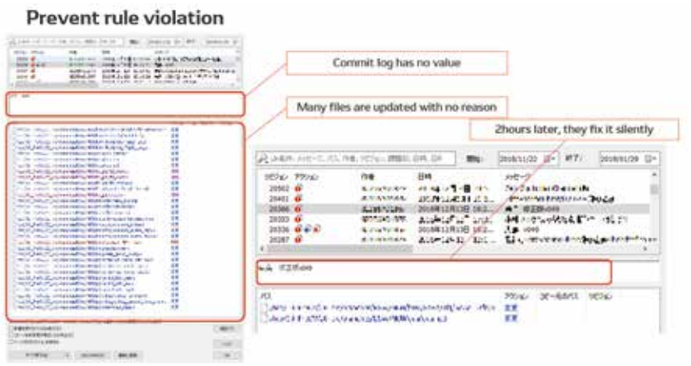

Fig. 3　Illegal workflow with their own policy

3. 開発者急増に伴う課題

開発者が急増することで，ソースコード管理ツールの使用方法に関しても新たな問題が発生していた．ソースコード管理ツールをまったく使用したことがない未経験の開発者であれば，手順書通りの作業を行うだろう．すでに経験があり熟練している開発者ほど自己流や前プロジェクトの使用方法を好む傾向があった．具体例として Fig. 3 に例を示すように，ローカルでソースコードを編集し，チェックアウトした環境に一括でファイルを上書きしてからコミットするというケースがある．しかし，この方法では無関係のファイルも一律に更新されてしまう．自身が更新したファイルのみを登録している訳ではないため，コメントもバージョン情報のみで何を変更したか明確化できていない．問題が発生したことに気づいて，あとから修正版をひそかに登録するということが発生していた．

プロジェクトとしてはルールや開発プロセスの教育を実施したつもりとなっていても，すべての開発者に実際に浸透するには時間がかかる．Subversion ではコミットを防ぐ手段が乏しく，ソースコード管理システム側で防ぐ手段が必要であった．

4. GitHub Enterprise Cloud を中心としたソフトウエア開発環境

ここで GitHub Enterprise Cloud について簡単に紹介する．GitHub Enterprise Cloud はクラウド上に提供される高度な Git サービスであり，ソフトウェア開発のライフサイクル全体を改善するための開発者プラットフォームである．Git のホスティングだけではなく，以下のような利点がある．

- ユーザー管理やアクセス制御など企業のニーズに合わせてカスタマイズすることが可能である．プラグインやアプリケーションとの連携が可能で，他の案件管理ツールやコミュニケーションツールと組み合わせて使用することができる．
- リポジトリを共有し，ソースコードの変更を追跡できる．プルリクエスト，コードレビュー機能や Git の blame 機能を活用することで，誰が・いつソースコードを修正したのかトレースすることができる．
- GitHub Actionsを使用してクラウドのコードベースでCI/CDのパイプラインを構築することができる．実装したソフトウェアのテストやデプロイなど定型業務を自動化できる．
- 企業向けの専用サポートサービスを提供している．英語でのサポート依頼であれば，課題発生時に優先度に応じて日本時間の夜間にもサポートを受けることができる．

コードレビュー・プルリクエスト・Issue 管理など効率的な開発プロセスをサポートしている．また，SAML 認証を用いて自身で準備した認証基盤を使用することができることも可能．これらの GitHub Enterprise Cloud の特徴を活用しつつ，弊社のセキュリティポリシーを担保し，前述の課題の解決を目指すこととした．

5. 各課題へのアプローチ

5.1 インフラ構築の期間短縮へのアプローチ

インフラ構築の期間短縮を実現するために，まずは必要なタスクの細分化を実施し短縮の目途について検討を行った．工程を大きく分けると 2 段階に分かれていた．

(1) 専用線接続のために必要なセキュリティの監査
(2) 専用線接続のための執務室，機材の手配，工事の実施

5.1.1 セキュリティ監査機関の短縮

デンソー社内に構築されているソースコード管理システムにアクセスするために専用線を開通すると，たとえファイアウォールで専用線経由のアクセス先を制限したとしても回線そのものはデンソーのネットワークにアクセスすることになるため，専用線開通に対するセキュリティ項目に対応する必要がある．例としてネットワーク分離や入退室の管理などの項目があげられる．これは本来実現したいソースコード管理システムへのチェック項目ではなく，デンソーのネットワークにアクセスする専用線に対して必要なチェック項目である．

チェックの対象がソースコード管理システムではなく，システムに接続するためのネットワークへの監査となっており，ここを改善のポイントとすることとした．そのためには，専用線に依存しないインフラが必要となる．

5.1.2 専用線接続の設備準備の短縮

GitHub Enterprise には大きく 2 種類のホスティング方法がある．1 つは GitHub Enterprise Server と呼ばれているもので，サーバーのソフトウェアを入手し，サーバーそのものは自前で構築する方法である．もう 1 つは GitHub Enterprise Cloud で，GitHub 社がクラウド上にホスティングしているサービスである．サーバーの自前構築を行う場合，デンソー社内に構築することになり，専用線が必要な点については解消することはできない．一方で，サーバー含めて自社管理となるため，アクセス制限やログのユーザー管理が柔軟に行えるのが利点である．GitHub Enterprise Cloud を選択した場合，インターネット接続ができる環境があればネットワークとしては接続できるようになるため，専用線に依存しないソースコード管理システムを構築することができる．

今回はインフラの構築期間の短縮を目的としているため，GitHub Enterprise Cloud を採用することとした．

5.1.3 インフラ構築の期間短縮試算

GitHub Enterprise Cloud を活用することでインターネットにアクセスでき，GitHub にアクセスできる環境であればソースコード管理システムにアクセスすることができる．デンソーの社内へのネットワークアクセスは不要となり，本来目的としていたソースコード管理システムに対するアクセスの監査へ変更することができる．

インターネットへのアクセスできない協力会社は現在ではほぼ存在せず，工事や敷設の期間はゼロにすることができる．

結果，インフラ構築までにかかる期間は半年以上かかっていた工程が 1 ～ 2 週間まで短縮することができる見込みが立った．

5.2 急増する開発者への対策

開発者が急増するにつれて，紛れ込むルール逸脱への対応も必要となっていた．自己流のソースコード管理システムの乱用に対しては，Fig. 4 に示すように GitHub の機能を使いプルリクエストによるレビューの実施と，ブランチ保護ルールによるプルリクエストレビューの必須化とする方針とした．すべての変更は自身のみでは完了せず，必ず第三者の目を通してから本線へマージされるフローとした．

5.3 アクセス制限とユーザー管理の課題

このようにメリットが多いように見えた GitHub Enterprise Cloud によるサービスだが，どうしてもクリアできない課題があった．

それは，ファイアウォールによるアクセス制限とユーザー管理の記録，アクセス記録保存ポリシーである．現在は GitHub Enterprise Cloud にも IP アドレス制限の機能が実装されているが，導入当時はその機能がなかった．GitHub Enterprise Server であれば上記

は実現できるが，インフラ構築の短縮を目的とするとGitHub Enterprise Cloud を採用したかった．

そこで Fig. 5 に示すように Identity Provider（IdP）を活用した認証の仕組みを用いてユーザー管理とアクセス制限を実施することにした．IdP とはユーザーの識別・認証情報の登録や記録，管理を行い，他のシステムやサービスに認証サービスを提供するシステムのことである．従来，ユーザー登録が必要なシステムでは，ユーザーの情報管理を各システムで行うのが一般的である．ユーザーの登録，ID の発行，パスワードなどの秘密の情報の保管，ログイン時のユーザー認証などの機能を各システムで実装・運用している．

しかし，ID フェデレーションという仕組みを用いることで認証に関する情報の管理や認証機能を外部の IdP に委ねることができる．GitHub Enterprise Cloud は SAML/SCIM といった仕組みを持っており，IdP で ID やパスワード，連絡先などの登録・管理を行い認証することができる．IdP で認証情報のアクセス記録の管理，アクセス元の管理を実施することで，GitHub Enterprise Cloud に欠けていたピースを埋めることができる．IP アドレス制限，アクセス記録の管理すべて満たせる見通しが立った．

6. 試験運用

前述のようにプロセス，ルールを定めて試験運用を開始したが，いくつか新たな課題も見えてきた．

1 つ目は Git そのものに不慣れな開発者も多く，特に Subversion と混同しがちという点である．同じ「コミット」という単語であっても，Subversion の「コミット」と Git の「コミット」では意味が異なる．また，分散リポジトリの仕組みも不慣れな人には複雑であり，ローカルリポジトリに Git「コミット」するだけで，リモートリポジトリに反映されているという勘違いも多かった．これに対しては Git の手順書を作成し，説明会を複数回開催，使い方で困ったときの問い合わせ窓口を設定することで対策した．

2 つ目はプルリクエストによるレビューが，集合レビューと重複して開発者の負荷になっていること．集合レビューで事前に確認しているため，プルリクエストのレビューが形骸化してしまう点があった．これに対してはコードレビューを集合開催しなくても GitHub 上のプルリクエストで代替するようにした．従来は会議時間がなかなか調整できずレビュー開催が遅れがちであったが，GitHub 上でソースコードを見ながらレビューできるようになるため，プルリクエストでレビューすることで時間を節約できるメリットがある．また，レビューアも自己の空いた時間でソースコードをレビューできるようになり，リーダーの会議時間の取り合いで開発が遅延することを防げる．

3 つ目はユーザー管理をそれぞれのシステムに登録するため，煩雑になること．これに対しては，IdP を Active Directory のユーザーと連携することとした．Active Directory によるユーザー追加・削除がそのまま IdP へのユーザー削除へ反映されることとなり，ユーザー管理の工数を大幅に削減することができた．

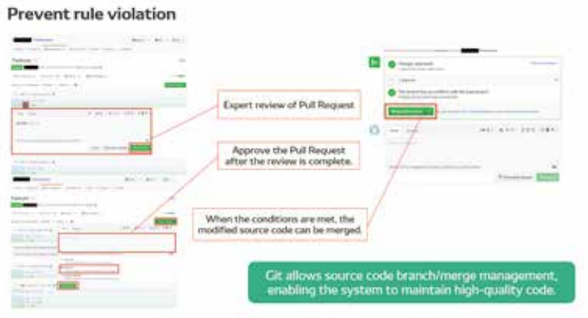

Fig. 4　Pull Request workflow of GitHub

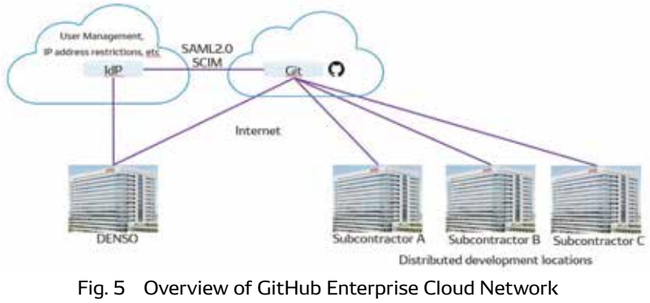

Fig. 5　Overview of GitHub Enterprise Cloud Network

7. インテグレーション環境の改善

GitHub Enterprise Cloud を活用することで，インフラ構築の短縮および急増する開発者の対応はできた．しかし，ソフトウェア開発者は慢性的に不足するようになってきており，ソースコード量の急増に対応するためには，開発効率そのものを向上させる必要がある．特にソースコード量に比例するようにインテグレーションの工数が増えていたため，インテグレーションの環境も改善する必要性があった．

GitHub では CI/CD ツールとして GitHub Actions という機能を備えている．

CI/CD は「Continuous Integration（継続的インテグレーション）／Continuous Deployment（継続的デプロイ）」の略称で，ソフトウェア開発プロセスの効率化において重要なファクターである．

CI は開発者が自動化されたプロセスを通じてソースコードをインテグレーションすることを指す．複数のエンジニアが個別に作業しているソースコードを定期的にビルドし，新しい変更を共通のリポジトリに取り込む．これにより，作成したソフトウェアに問題がないかをテストし，ソフトウェアの不具合を早期に見つけることができる．

CD は継続的にソフトウェアを提供することを指す．今回の構築した環境においては，最新版のソフトウェアをライブラリ化し，他の開発者に必要なライブラリを継続的に提供し続けることを対象とする．

CI/CD ツールの代表的な例として Jenkins というフリーソフトがあるが，コミュニティの盛んな伝統的なツールであるが故に，以下のような問題があった．

(1) ジョブはセルフホストで実行されるため，例えばビルドエラーが発生した場合，ユーザーが原因を解析するためにはセルフホストにアクセスできなければならない．そのためセルフホストにアクセスできる環境を準備する必要が発生する．

(2) ジョブに実行環境の定義をすることが難しく，例えばホストにインストールされているコンパイラ，SDK（Software Development Kit）や静的解析ツールのバージョンによって結果が異なってしまい，冪等性を担保できないことがある．

(3) 様々なプラグインが開発されているが，プラグインのバージョンアップは開発者の善意に委ねられている．また記述の自由度が非常に高く，職人問題が発生しやすい．

筆者も古くからの Jenkins ファンであり，Jenkins がカスタマイズ性に長けた有用な CI ツールであることは理解しているが，カスタマイズ性に長けるということは開発者による独自拡張（逸脱）を許容することに他ならない．5.2 節で述べたように，急増する開発者に対応するためには，独自の運用がなるべくできないようにする必要がある．そのため，今回の環境においては実行環境を含めて定義し，それを Git 上で構成管理できる GitHub Actions を主軸とすることとした．

8. GitHub Actions を活用した環境構成

Fig. 6 に今回構築した環境構成の概要を示す．しかし，これはあくまで骨子となる仕組みであり，日々改善を行っている開発環境では既に最新のものではなくなっていることはご容赦願いたい．

Fig. 6 の (1) は Build Check の環境である．開発者はソースコードのプルリクエスト時にビルドチェックが自動的に行われ，ビルドチェックに通過したソフトウェアしかマージができない仕組みとしている．コンパイラやリアルタイム OS の情報は Fig. 7 に示す JSON ファイルに定義し，JSON ファイルを元に Fig. 8 で Matrix Build の生成する．コンパイラやリアルタイム OS ごとに並列してビルドが実行されるため，短期間

でビルドチェックを行うことができる．

Fig. 6 の (2) は外部コラボレータとのデータの同期である．デンソーから委託開発している Organization（GitHub では Organization 単位で参加メンバーを管理することができる）とは異なる Organization を準備し，外部コラボレーターとの協力専用の環境を準備している．社外の協力者がアクセスする環境と社内の開発用の環境を完全に分離した上で，リポジトリの同期処理を GitHub Actions を活用して自動化している．

Fig. 6 の (3) はリリース候補の作成と成果物のデプロイを示している．多数のリポジトリのバージョン管理をするため，Android で使用されている repo というツール [2] を活用している．ソースコードが格納されたそれぞれのリポジトリのバージョン情報を Build Manifest として XML ファイルに定義している．コンパイラとリアルタイム OS については前述の通り JSON ファイルで定義をし，Matrix Build を生成している．生成される成果物は 10GB を超えるサイズとなるため，セルフホステッドランナーを採用しているが，冪等性を担保するためビルドに必要な環境を Dockerfile で定義し，Docker 内でビルドを実施している．ビルドした成果物を Jfrog Artifactory に登録され，実行したジョブと紐づけて管理される．GitHub Actions の Marketplace には有用な公式 Action が登録されており，このような連携を短期間で実現できる．

これらの環境により，あまりにソースコードが多くビルドを通すのに 2 週間かかっていたインテグレーション環境が，ビルドエラーが発生せず評価に工数を割くことができるようになった．

```
{
 "include":
 [
  {
   "artifact":"ArtifactName",
   "osver":"v5.2.3",
   "compiler":"v2.1.1",
   "libver":"develop",
   "buildmode":"debug"
  }
 ]
}
```
Examples of definitions

Fig.7　Definition file of environment

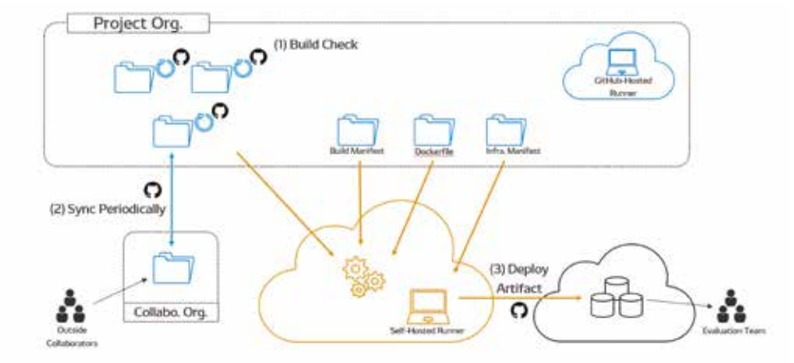

Fig. 6　Overview architecture of CI/CD pipeline

```
jobs:
 SetupMatrix:
  runs-on: ubuntu-18.04
  outputs:
   matrix: ${{ steps.set-matrix.outputs.matrix }}
  steps:
  - name: Checkout Config File
   uses: actions/checkout@v2
   with:
    repository: *****/***-build
    ref: master
 - name: Parse ***-Build Config
   id: set-matrix
   run: echo ::set-output name=matrix::$(jq -c "" < .github/config/***-build.json)
```

Matrix Build Sections

Parse the definition file

- The version of cross-compiler
- The version of RTOS
- The versions of dependency libraries
- Build mode

...etc

Fig. 8　Mesh Concatenation (Traversing)

むすび

ツールチェーンを活用した特定の人に依存しないソフトウェア開発環境の達成が理想である．

ソフトウェア開発環境をクラウド化し，開発者の増減に柔軟に対応する環境を準備できた．さらにプルリクエストを活用したレビュー，インテグレーションの自動化，冪等性の担保を達成した．更なる開発の効率化に向けて，自動テスト環境やミドルウェアの共通化も準備している．

車載のソフトウェアがより複雑化し，OTA によるアップデートも必要になっている中，目指す姿は DevOps 環境の構築である．ソフトウェアの実装からインテグレーション，自動テストまで含め，更なる環境改善を進めていくき，ソフトウェアの更なる効率化を目指していく必要がある．

参考文献

1)「NXP S32x 車載プロセッシング・プラットフォーム」NXP Semiconductors(2017/10)
2) Repo コマンドリファレンス https://source.android.com/docs/setup/create/repo

著者

望月 洋二
もちづき ようじ
ソフトウェア技術1部
Microsoft Certified: DevOps Engineer Expert
AD／ADAS 向けのソフトウェアプラットフォームの開発に従事