Einblicke in die Softwareentwicklung für autonomes Fahren und Sensorfusion im Automobilbereich unter Beachtung der Anforderungen an Verlässlichkeit sowie einfache und leichte Wartung

Qualitätskriterien für gute Software - Entwicklungsprozess und Lebenszyklus

Maritimes Cluster Norddeutschland
Oldenburg / 31.01.2024

Dr. Michailas Romanovas

We transform automotive mobility

CARIAD
A VOLKSWAGEN GROUP COMPANY

Haftungsausschluss

- Die geäußerten Ansichten und Meinungen sind ausschließlich die des Autors und stellen keine offizielle Meinung der CARIAD SE oder eine durchgesetzte Richtlinie

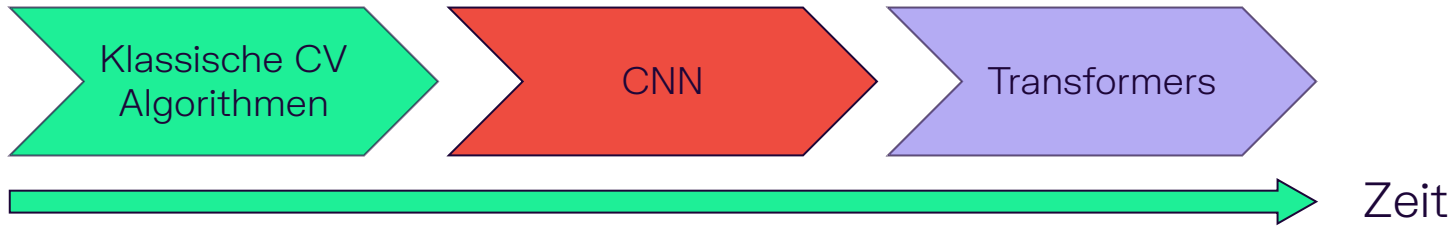
- Die Informationen basiert auf den Berufserfahrungen des Autors, die er sowohl vor und bei CARIAD SE in Automobileindustrie als auch während seiner Tätigkeit in der öffentlichen Forschung gesammelt hat

- Die Information ist mit Vorsicht zu genießen:
  - Jedes Projekt oder jede Initiative eine einzigartige Kombination von Anforderungen haben kann und je nach Projektgröße, Tools, Budget und Hintergrund der Entwickler unterschiedliche Tools oder Prozesse implementiert werden

Autonomes Fahren

- Autonomes Fahren sieht komplexer zu sein als ursprünglich erwartet
- Die Diskussion erfolgt aus der Perspektive der Entwicklung von Wahrnehmungsalgorithmen
  - Sensorfusion, Freiraumschätzung, Multiobjektverfolgung,...
- Wahrnehmung als Eingangsschritt für die Kette sicherheitskritischer Funktionalität
  - Außergewöhnliche Herausforderungen aufgrund unstrukturierter Daten und Informationsmengen

- Schnelle algorithmische Entwicklung im letzten Jahrzehnt: Paradigmenwechsel:
  - KI bis zu **End-to-End-Fahrfunktionen**



- Erleichtert durch die Erhöhung der verfügbaren Rechenleistung an Bord des Fahrzeugs

Technische Qualität der Softwareentwicklung

Kontext:

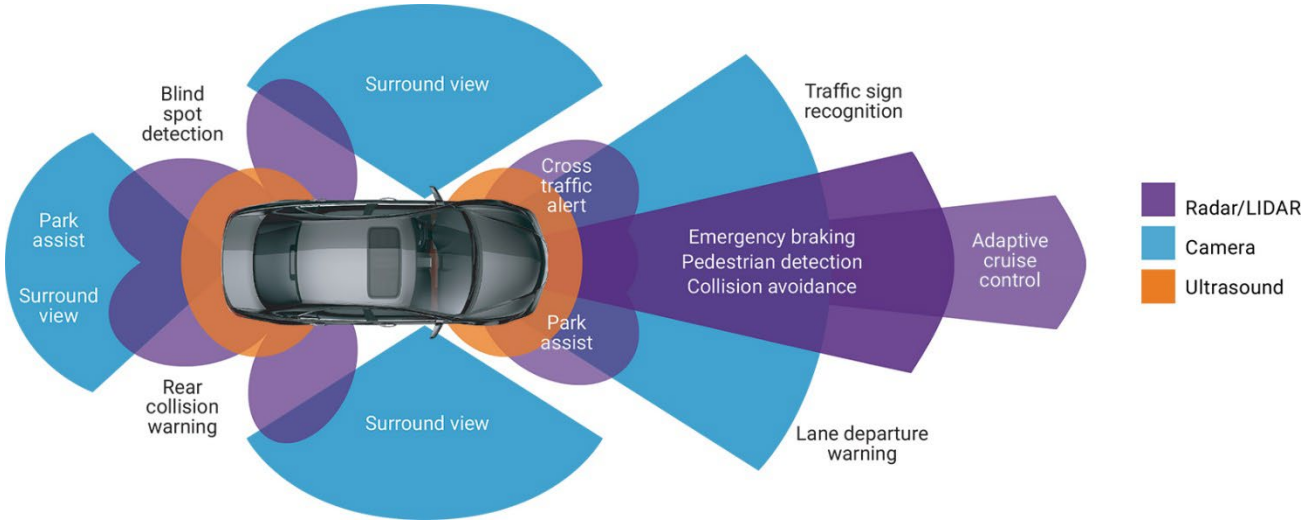
- Autonomes Fahren/Umweltwahrnehmung
- Sensorfusion und Objektverfolgung

Fragen:

- Best Practices zur Softwarezuverlässigkeit und Wartbarkeit in sicherheitskritischen Anwendungen

Worüber wir reden:

- ToDo's und praktische Tipps!
- Die allerersten Schritte
- Akzeptanz von den Entwicklern
- Beispiele für C++ als Entwicklungssprache



Hypothetischer Sensoraufbau für ADAS-Anwendungen (courtesy of embedded.com [link]

ASPICE / V-Model / ISO 26262 / SOTIF

ASPICE (Automotive Software Process Improvement Capability Determination)

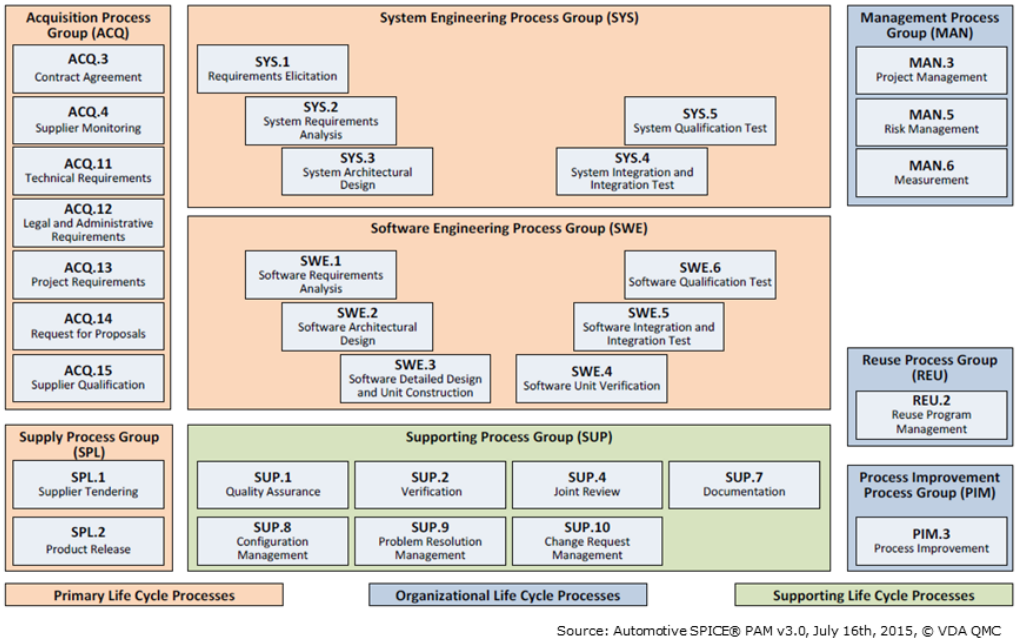
- Richtlinien zur Bewertung des Softwareentwicklungsprozesses
- Domänenspezifische Anpassung des ISO 33061-Standards
- Best Practices zur frühzeitigen Erkennung von Defekten

ISO 26262:

- Komplementär zu ASPICE mit einigen Überschneidungen
- Internationaler funktionaler Sicherheitsstandard für die Entwicklung von E/E-Systemen in Straßenfahrzeugen
- Sicherheitsstandard mit steigenden Anforderungen je nach ASIL-Level mit Fokus auf technische Praktiken → ASPICE erwähnt nur generische Anforderungen / Rückverfolgbarkeit

SOTIF (Safety Of The Intended Functionality)

- Baut sich auf ISO 26262
- Stellt sicher, dass die erforderliche Funktionalität unter unbekannten Bedingungen und ohne Auftreten eines Fehlers gewährleistet werden kann.



Source: Automotive SPICE® PAM v3.0, July 16th, 2015, © VDA QMC

Die „V"-Form liegt daran, dass die Test- und Verifizierungsschritte in umgekehrter Reihenfolge von Entwurf und Implementierung durchgeführt werden.

Anforderungen

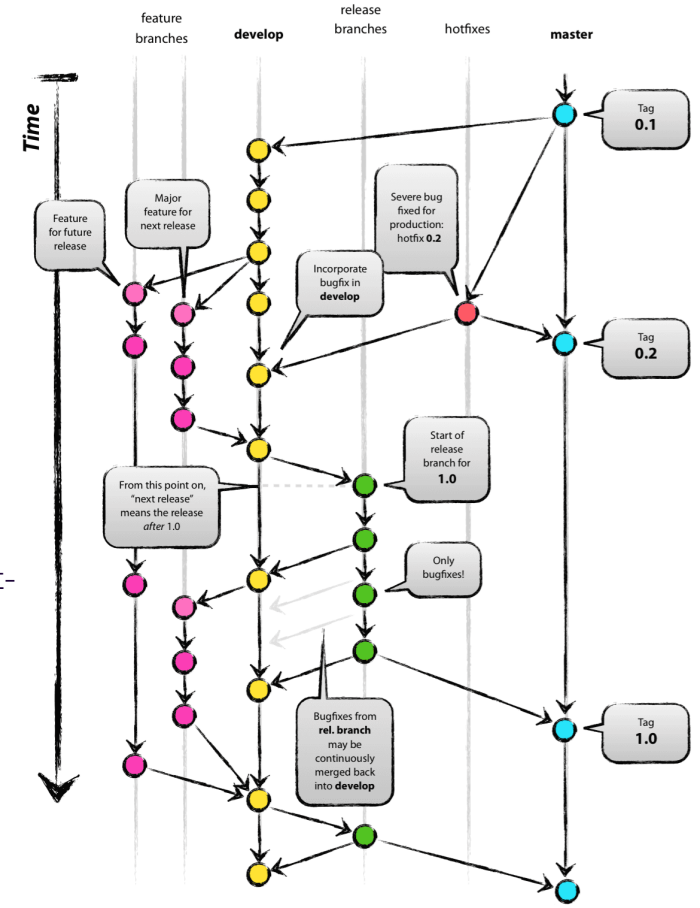
- Anforderungserhebung und als kritischer Teil des ASPICE-Prozesses
  - Gut etablierter Prozess in der Softwareentwicklung
  - Viel Know-how, das nicht automobilspezifisch ist
- Im ersten Schritt können klassische Verfahren grundsätzlich strenger und systematischer angewendet werden
  - Beachten und befolgen ISO 26262
  - Funktionales Sicherheitskonzept und Sicherheitsziele

Wichtige praktische Überlegungen:

- Rückverfolgbarkeitskonzept (Traceability): Sicherheitsziele ↔ Anforderungen ↔ Design
- Fehleranalyse und -klassifizierung zur Bestimmung von ASIL-Level, FMEA usw.
- Anforderungsänderungen
- Technische Mittel: externes (Codebeamer) vs. „Nebencode“
  - Der Konsistenz der Anforderungen und der Codebasis (Abdeckung usw.)

Versionskontrollsystem: GIT

- Entscheidende Rolle bei der sicherheitskritischen Softwareentwicklung
- Ermöglicher oder Unterstützer von:
  - Rückverfolgbarkeit und Verantwortlichkeit
  - Fehlererkennung und Fehlerbehebung
  - Kollaborative/Parallel Entwicklung und Verzweigung
  - Codeüberprüfung und Qualitätssicherung
  - Konfigurations-/Release-Management
  - Auditierung und Compliance
  - Reproduzierbarkeit und Rollback
  - …
- Vereinbaren von Anfang an die GIT-Richtlinien:
  - Verzweigungs- und Release-Strategien
  - Behandlung von Commits → erfordert möglicherweise Disziplin von den Entwicklern



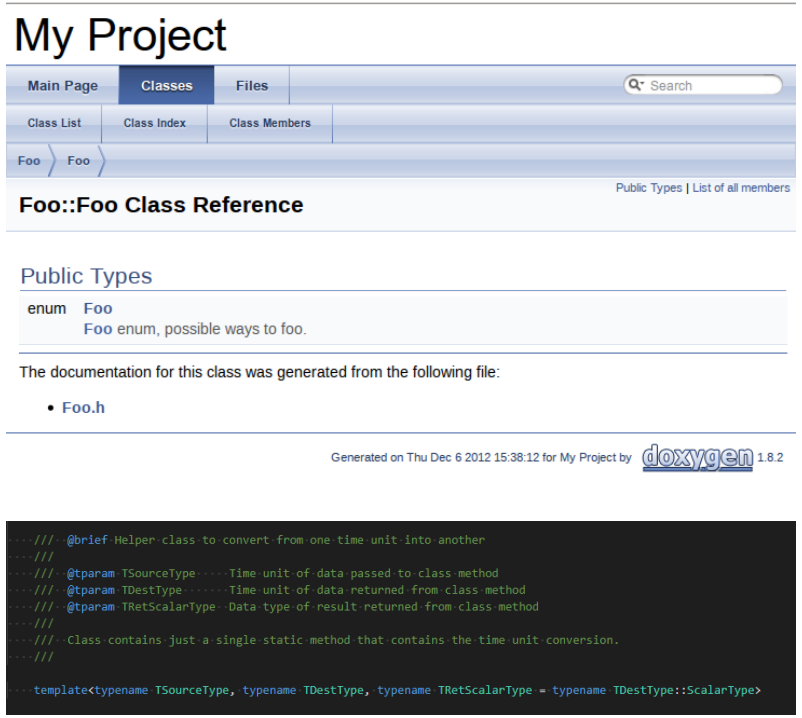
Ein erfolgreiches Git-Branching-Modell (courtesy: nvie.com)

Entscheiden ob ein einzelnes (auch Mono-Repo genannt) oder mehrere Repositories mit Paketverwaltung (z. B. mit Conan) verwendet

Eine technische Entscheidung, die stark von zahlreichen Faktoren abhängig ist!

Quellcode-Dokumentation

- Mit Tools (z.B. Doxygen für C/C++) als wichtiger Bestandteil für die sicherheitskritische Softwareentwicklung
- Ermöglicher oder Unterstützer von:
  - Nachweis der Einhaltung von Sicherheitsstandards/-vorschriften
  - Überprüfungsprozess und Inspektion
  - Wartung und zukünftige Entwicklung
  - Codeverständnis und Wissenstransfer
- Dokumentation bleibt mit dem Quellcode synchronisiert und überprüft!
  - Der Prüfer prüft, ob die Dokumentation (bis zu einem gewissen Grad) mit der Implementierung übereinstimmt.
- Integration als Teil der CI/CD-Kette → Konsistenz von Software-Releases!
- Inklusive Zitierung wissenschaftlicher Arbeiten und LaTeX-Gleichungen für mathelastige Themen
- Betrachtung von Dokumentationsfehler als Quellcodefehler!
- Kann als Ersatz für das detaillierte Design dienen



Coding Richtlinien

- Legen/vereinbaren frühzeitig Namens- und Formatierungskonventionen fest
  - Benutzerdefinierte Formatierungsstile vs. vordefinierte Codeformatierungsstile halten (z. B. Google, LLVM usw.)

- Ermöglicher oder Unterstützer von Folgendem:
  - Konsistenz des Codes → verbesserte Bereitschaft der Entwickler zur Zusammenarbeit an der Codebasis
  - Reduziertes Risiko häufiger Programmierfehler → verbesserte Sicherheit und Zuverlässigkeit des Codes
  - Verbesserte Lesbarkeit und Codeüberprüfungsprozess → erhöhte Effektivität
  - Verbesserte Wartung und Updates → geringeres Risiko, Fehler bei Codeänderungen
  - Unterstützt durch z.B. Linter-Tools,
    - Vermeiden Verstöße als Fehler zu betrachten

Statische Code-Analyse (SCA)

- Analyse der Code-Konformität, ohne ihn auszuführen: Vermeidung von undefiniertes / nicht spezifiziertes Verhalten
  - MISRA 2008/2023 für C++ (typische Kundenanforderung in Automotive)
- Verwenden ergänzende Tools, sofern verfügbar (z. B. AUTOSAR C++14, CERT CPP-Regeln)
- Beachten auf Fehlalarme
  - SCA-Tools sollen den Code besser und nicht schlechter machen
  - Stellen sicher, dass die Tools neuere Sprachstandards echt unterstützen (z. B. MISRA C++2008 vs. C++17)
- Nicht alle Regeln blind verfolgen: wählen nur relevante Regeln aus
  - Trennung sicherheitsrelevante Regeln von „Codierungsrichtlinien“ (Design, Wartbarkeitsregeln)
  - Abweichungen vom Regelwerk und Regelverstöße im Code sind zu dokumentieren und zu begründen
  - Erwägen das Hinzufügen kundenspezifischer/projektspezifischer Regeln
- Verwenden zertifizierte Tools zur Code-Compliance-Prüfung: Verhindert schlechten Code (aber fördert keinen guten Code!)
- SCA muss Teil der CI/CD-Kette sein → die Anzahl der zulässigen SCA-Warnungen begrenzen und zu verringern (auch Regeln anpassen!)

Dynamische Code-Analyse

- Tools zur Identifizierung von Schwachstellen im Laufzeitverhalten
  - Erkennung von Speicherbeschädigung, Speicher leaks, nicht initialisiertem Zugriff, Parallelitätsproblemen, usw.
- Ergänzung zur statischen Code-Analyse - kann zur Kreuzvalidierung von SCA-Ergebnissen verwendet werden
  - Werkzeuge sind komplexer
  - Die genaue Ortsverfolgung der Schwachstelle ist nicht so einfach wie bei SCA
- Einige gute Tools wie Valgrind und Google Sanitizers

WARNUNG:

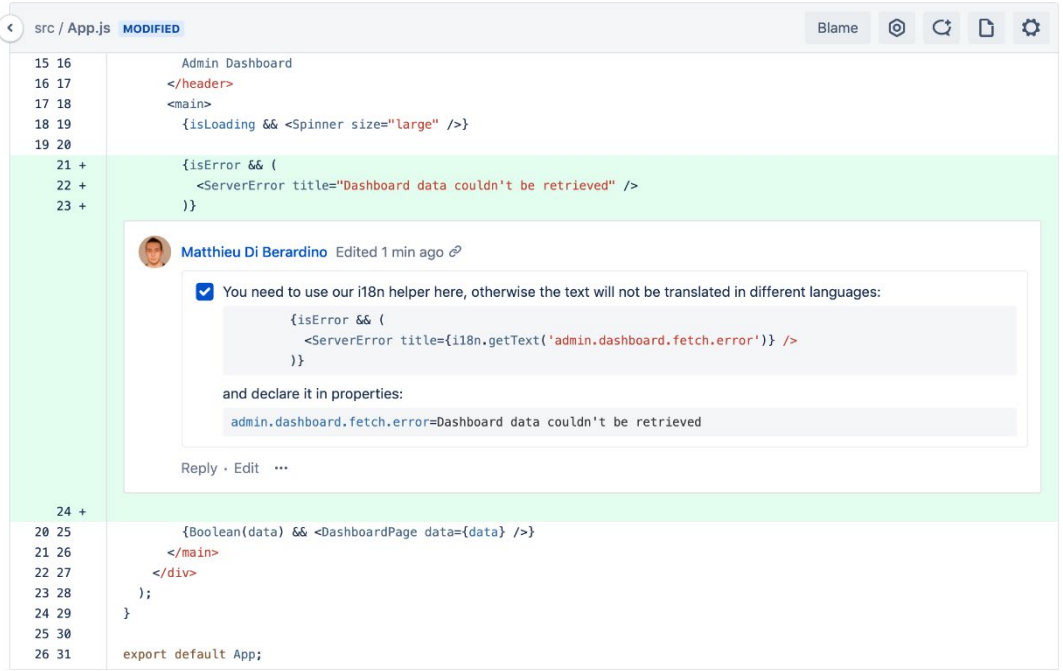
- Zusammen mit SCA kann ein falsches Sicherheitsgefühl vermittelt werden
  - Die dynamische Analyse findet Probleme im ausgeführten Code → die vollständige Abdeckung möglicherweise immer noch ein Problem ist

Peer Review und Codeeigentum

- Peer Review: äußerst wichtiger Schritt
  - Ohne PR geht einfach nicht
  - Erfordert von den Entwicklern Engagement zur Zusammenarbeit
  - Dient auch der Wissensvermittlung

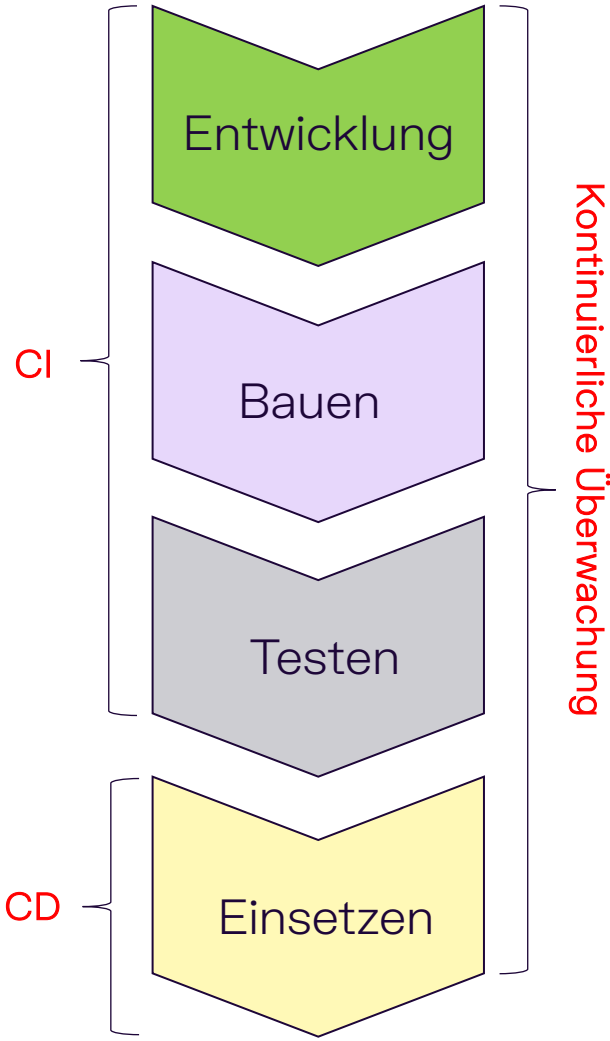
- Sehr wichtig ist die Verwendung des Code-Eigentums
  - Leserschaftskonzept (Kompetenzakkreditierung)
  - Expertenmeinung zum relevanten Code
  - Wichtig für Großprojekte

- Es darf nur „zertifizierter“ externer Code verwendet werden
  - Z. B. eine Teilmenge der STL, die vom Plattform-/Compiler-Anbieter zertifiziert werden muss



CI/CD Kette

- Entscheidender Bedeutung für die Bereitstellung sicherheitsrelevanter Software
- Der Entwickler muss umgehend Feedback erhalten
  - Frühzeitige Erkennung von Fehlern / Schnelle Feedbackschleife
- Beinhaltet:
  - Verschiedene Bauen-Schritte/Konfigurationen (alle relevanten Setups)
  - Code-Analyse (statische, dynamische, benutzerdefinierte Codierungsrichtlinien)
  - Ausführen von Tests (Unit-Tests, Integrationstests,…) und KPIs
  - Embedded deployment
  - Dokumentation, Berichterstattung
  - Generierung von Artefakten und Softwarebereitstellung
  - Erstellen Kennzahlen zur Codequalität, KPIs. → Langzeit-Leistungsbeurteilung
- Die Einhaltung von Sicherheitsstandards für die Prüfung von Änderungen und die Validierung des Entwicklungsprozesses
  - Compliance und Auditing (transparente und überprüfbare Nachverfolgung)
- Erleichtert den PR-Prozess → Kollegen überprüfen nur den Code, der die CI/CD-Schritte erfolgreich bestanden hat



CI/CD Kette: Einige Kommentare

- Bei großen Projekten und hohen Qualitätsanforderungen kann sehr lange dauern
  - Konfliktlösung wird eventuell ein Thema

Wichtig: Neben dem Produktionscode kann die Entwicklung, Aktualisierung und Wartung der CI/CD-Kette zu einem der größten Teile der Entwicklungsarbeit werden

- Hinweise:
  - Erstellen eine CI/CD-Kette so früh wie möglich
    - Beheben die technischen Schulden so früh wie möglich (z. B. semantische Konflikte, Laufzeit usw.).
  - Erwägen die Ausführung auf eingebetteten Zielsystemen als Teil der CI/CD-Kette

CI/CD Kette: Bauen-Prozess

- Nutzen das komplementäre Verhalten verschiedener Compiler
  - Einsetzen z.B. für C++: GCC / Clang / MSVC / …
- Verwenden unterschiedliche aber relevante Konfigurationen
  - Debug vs. Release (z.B. um dynamische Asserts und aggressive Optimierungen zu testen)
- Weitere Konfigurationen können Schritt für Schritt hinzugefügt
- Maximalen Satz relevanter Warnungen/Fehler aktivieren
- Behandeln von Warnungen als Fehler

- Überwachen die CI/CD-Leistung und etablieren Sie „Gesundheitsmetriken“ (z. B. DevOps Research and Assessment – DORA).
  - Change failure rate and mean time to repair

Testen (Einheit, Integration, Smoke, …)

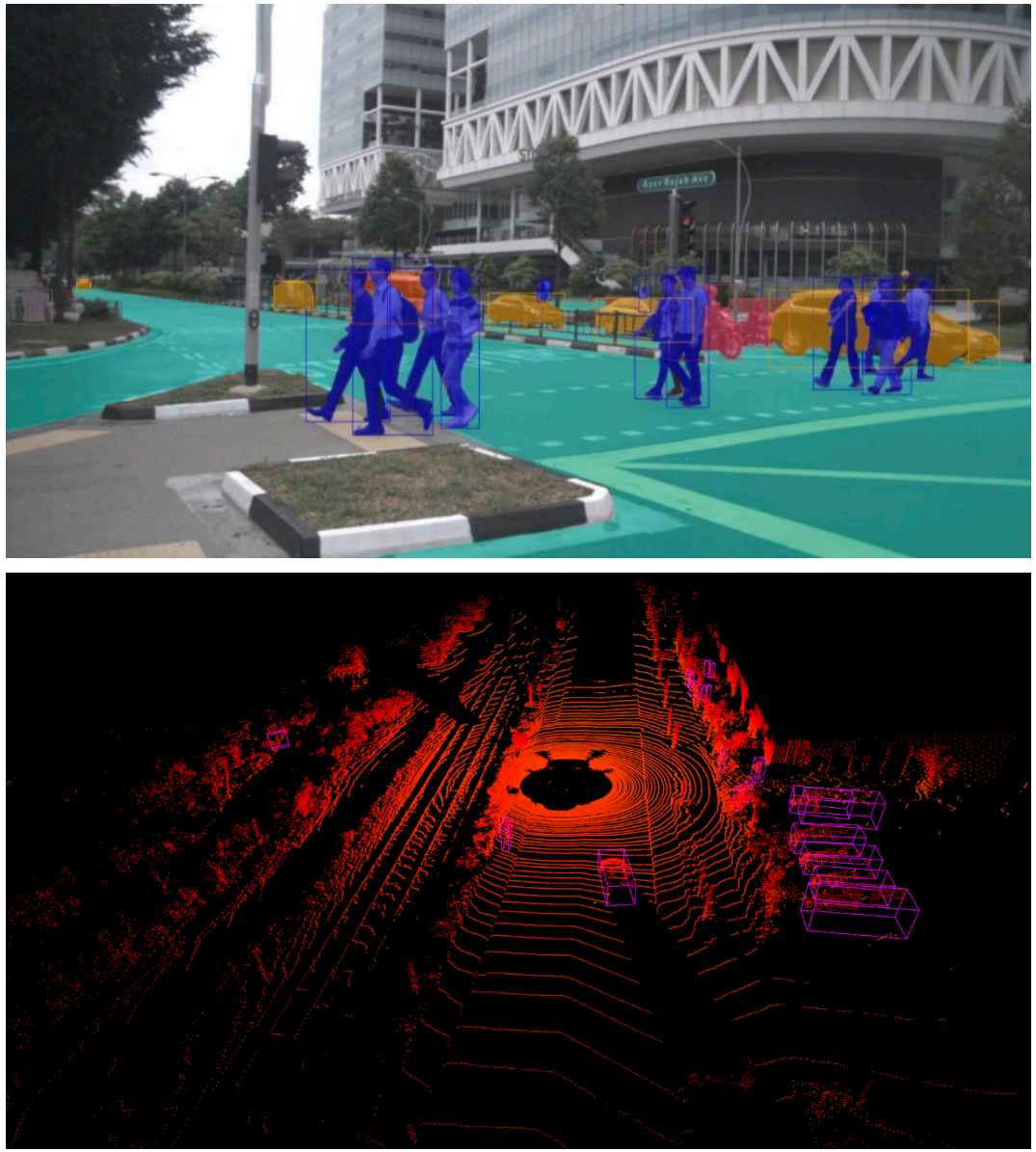
- Unbestreitbare Bedeutung für sicherheitskritische Softwareentwicklung: Teil des V-Modells
- Abdecken alle Teststufen inkl. Funktionstests
  - Unit- und Integrationstests müssen frühzeitig in die CI/CD-Kette übernommen werden
- Stellen sicher, dass das Schreiben von Unit-Tests ein wesentlicher Bestandteil der Entwicklung ist
  - Teil von DoD
- Entscheiden auf welcher Ebene Tests von der Entwicklung getrennt werden
  - Ist der Entwickler für den Unit-Test, den Integrationstest verantwortlich?

Hinweise

- Es reicht nicht aus, eine Line/Branch-abdeckung von 100 % zu erreichen:
  - Ggf. entsprechende Äquivalenz Klassentests usw.
- Nachvollziehbarkeit der Tests gegenüber Anforderungen und Design: Dokumentation der Tests
  - Für mathelastige Tests: woher die Referenzwerte kommen?
- Ein gutes und zertifizierbares Test-Framework (z.B. GTest) mit Mocking-Unterstützung
  - Selbst schreiben?

Performance KPIs

- Frage: führt meine Funktionalität zu einer besseren Endleistung des Systems?
  - Ziel: algorithmische Entwicklung
- Formulieren funktionsrelevante KPIs
- Erstellung von Testszenarien anhand von Referenzdaten/Ground Truth und kontinuierliche Überwachung die KPIs
- Daten Quellen:
  - Simulierte Daten
  - Manuell annotierte Daten
  - Automatisch erstellte Referenzdaten (bessere Sensoreinrichtung oder Offline-Algorithmen) → z. B. LiDAR



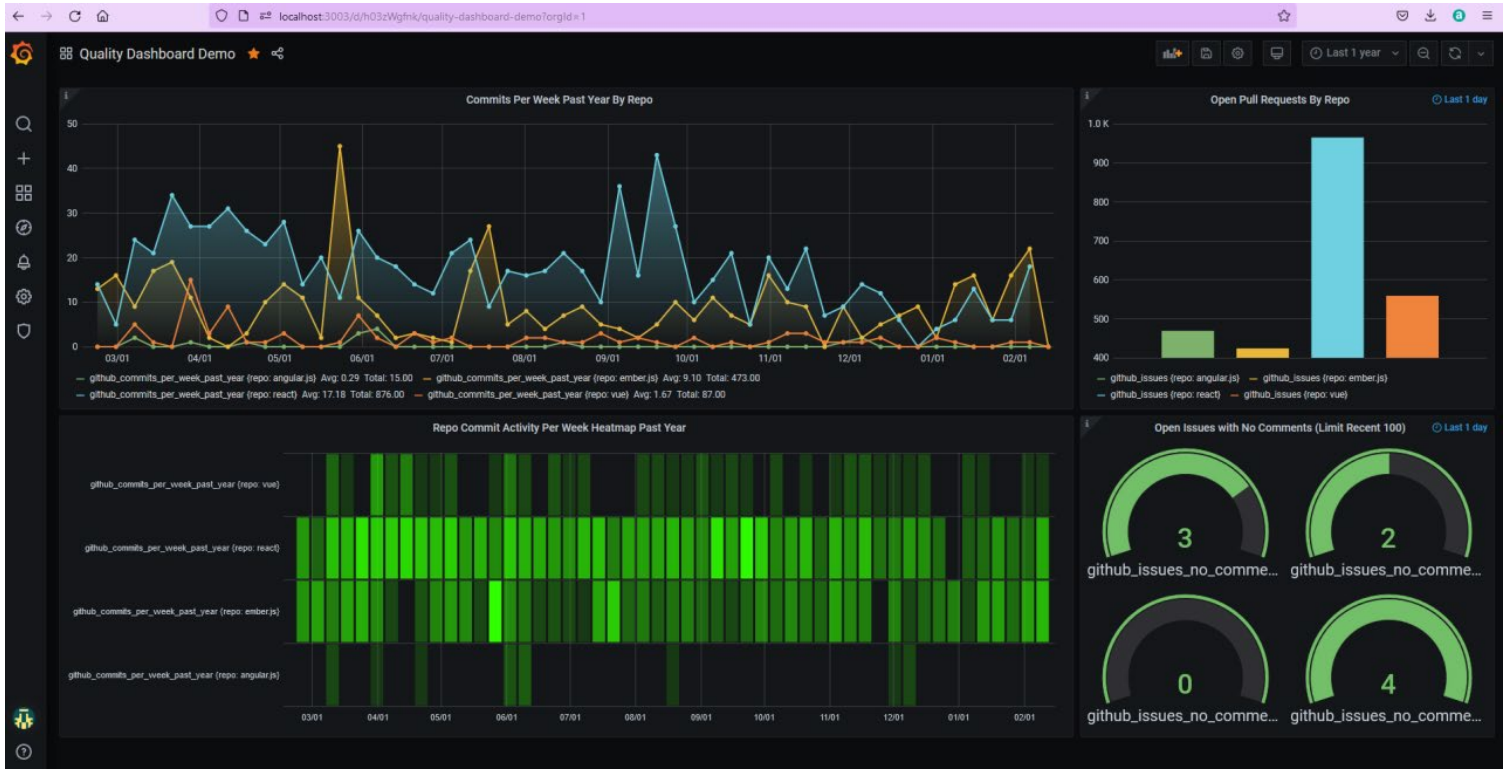
Ein Beispiel für den nuScenes-Datensatz und die LiDAR-Referenzdaten

Visualisierung der KPIs / Artefakte

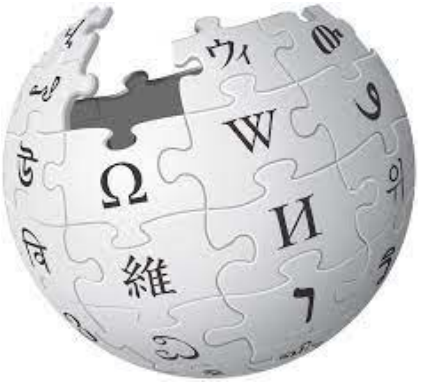
- Verschaffung einen umfassenden Überblick über den Entwicklungsprozess

- Nutzen die Observability-Plattform, um Fortschritt/Leistung zu visualisieren
  - Bei nächtlichen Läufen
- Zur Visualisierung im Zeitverlauf:
  - Speicher-/CPU-Auslastung
  - Gesundheit der CI/CD-Kette
  - Aktivität/Commits
  - SCA-Warnungen
  - Abdeckung
  - Algorithmische KPIs
  - …

Ein Beispiel für das Grafana-Board, das für die Leistungsvisualisierung verwendet wird

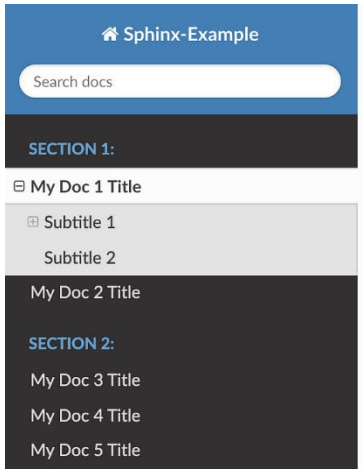


Projekt-/produktweite Dokumentation



- Größere Projekte erfordern möglicherweise eine eigene Dokumentation

- Enthalten: alles, was nicht in Code steht
  - Entwicklungsumgebung: Setup, Tools, Konfigurationen
  - Nutzungstutorials und hinweise
  - Architektur und Design
  - Anforderungen, Annahmen usw.
  - Sicherheitskonzept
  - Abweichungen von den SCA-Regelsätzen
  - Glossary
  - …



Hinweis

- Dokumentation aus dem Quellcode erstellen: Teil des Review-Prozesses zusammen mit dem Quellcode

Zusammenfassung

- Die Entwicklung der Software für ADAS ist ein anspruchsvoller Prozess
  - Algorithmische/funktionale Komplexität x die Größe des sicherheitsrelevanten Codes
- Allgemeiner Ablauf: ASPICE, ISO 26262 / SOTIF und V-Modell mit Anpassungen
- Zu mitnehmen
  - Anforderungsengineering
  - Git mit etablierten Prozessen
  - Quellcode-Dokumentation
  - Kodierungsrichtlinien (einschließlich Tooling)
  - Statische/dynamische Codeanalyse
  - Inkrementeller Aufbau und Pflege der CI/CD-Kette
  - PR Code-Ownership
  - Tests und Leistungs-KPIs
  - Visualisierung der KPIs/Artefakte
  - Projektweite Dokumentation

Vielen Dank!

Fragen?