Integration von Performability-Analysen in eine CI/CD-Pipeline

Vincenzo Pace

Institute of Information Security and Dependability (KASTEL)
Advisor: M.Sc. Sebastian Weber

1 Einleitung

Performability-Analysen sind Methoden, die sowohl die Leistungsfähigkeit als auch die Zuverlässigkeit von Systemen bewerten. Sie ermöglichen es uns zu verstehen, wie ein System unter verschiedenen Bedingungen arbeitet, einschließlich normaler Betriebszustände und potenzieller Fehler oder Ausfälle. Diese Analysen sind entscheidend, weil sie uns helfen, vorherzusagen, wie sich ein System in realen Umgebungen verhält. Durch das Verständnis dieses Verhaltens können wir Systeme entwerfen und optimieren, die nicht nur effizient, sondern auch robust und zuverlässig sind.

Warum ist das wichtig? In der heutigen Zeit sind Systeme oft komplex und müssen hohen Anforderungen gerecht werden. Indem wir Performability-Analysen durchführen, können wir Schwachstellen identifizieren, bevor sie zu echten Problemen werden. Das bedeutet, wir können proaktiv handeln, anstatt nur auf auftretende Fehler zu reagieren. So stellen wir sicher, dass das System unter verschiedenen Lasten und Bedingungen stabil bleibt.

Eine wesentliche Grundlage für diese Analysen sind Laufzeitdaten wie Traces und Metriken. Traces sind detaillierte Aufzeichnungen der Abläufe innerhalb eines Systems, während Metriken quantitative Messungen wie Antwortzeiten oder Ressourcenauslastung darstellen. Diese Daten werden von einem laufenden System gesammelt und liefern wertvolle Einblicke in dessen aktuelles Verhalten. Mit diesen Informationen können wir Modelle erstellen oder Simulationen durchführen, um zu prognostizieren, wie sich das System unter veränderten Bedingungen verhalten würde—beispielsweise bei höherer Benutzerlast oder bei Ausfall bestimmter Komponenten.

Um solche Analysen effektiv und kontinuierlich durchführen zu können, benötigen wir eine umfangreiche Software-Infrastruktur. Hier kommt die Automatisierung ins Spiel. Prozesse wie das Sammeln von Daten, deren Verarbeitung und die Durchführung der Analysen müssen nahtlos und ohne manuelles Eingreifen ablaufen. Eine CI/CD-Pipeline (Continuous Integration/Continuous Deployment) ist dafür die ideale Lösung. Sie automatisiert die Integration neuer Daten und die Bereitstellung der Analyseprozesse. Durch

den Einsatz einer CI/CD-Pipeline können wir sicherstellen, dass unsere Performability-Analysen stets aktuell sind und schnell auf Veränderungen reagieren.

Zusammengefasst helfen uns Performability-Analysen dabei, das Verhalten eines Systems unter verschiedenen Bedingungen vorherzusagen. Sie sind ein unverzichtbares Werkzeug, um Systeme zu entwickeln, die sowohl leistungsfähig als auch zuverlässig sind. Durch den Einsatz automatisierter Prozesse können wir diese Analysen effizienter gestalten und bessere Einblicke gewinnen, was letztlich zu robusteren und effizienteren Systemen führt.

Die Hauptaufgabe dieses Projekts bestand darin, eine CI/CD-Pipeline zu entwickeln, die es ermöglicht, Performability-Modelle kontinuierlich zu generieren und zu analysieren. Für diese Generierung und Analyse werden der Performance Model eXtractor von der Uni Würzburg [1] und der Palladio Simulator vom KIT [2] verwendet. Der Fokus lag auf der Integration dieser Software in die Pipeline, um eine automatisierte, kontinuierliche Analyse zu ermöglichen. Dazu musste ich zuerst die Umgebung des FZI für mich replizieren, um damit arbeiten zu können.

2 Problemstellung

Die Vision des Projekts war es, eine Umgebung für die kontinuierliche Performability-Analyse zu schaffen. Dabei werden die Traces und Metriken nicht durch ein Überwachungssystem bereitgestellt, sondern vom Nutzer als Teil des Repositories hochgeladen. Sobald der Nutzer neue Traces in das Repository pusht, wird der CI/CD-Prozess automatisch angestoßen. Die Pipeline übernimmt dann die Generierung und Simulation der Performability-Modelle basierend auf diesen neuen Daten. Die Ergebnisse der Analysen werden zentral im Repository gespeichert und stehen somit für weitere Auswertungen und Optimierungen zur Verfügung.

3 Lösungsansatz

Von Beginn an stand für mich fest, dass NixOS als Betriebssystem für die Replikation des FZI-Setups zum Einsatz kommen würde. NixOS bietet eine einfache und flexible Konfigurationsmöglichkeit, die es mir ermöglichte, die verschiedenen benötigten Komponenten, wie Docker und GitLab, schnell und effizient einzurichten. Die Entscheidung für NixOS erleichterte es mir, eine stabile und reproduzierbare Umgebung zu schaffen, in der alle Komponenten nahtlos zusammenarbeiten.

Durch meinen Betreuer erfuhr ich, dass das FZI statt der GitLab-Container-Registry die standardmäßige Docker-Registry verwendet, welche ich deshalb ebenso aufsetzte.

Um eine klare Trennung der einzelnen Dienste zu gewährleisten, machte ich die verschiedenen Komponenten über unterschiedliche Subdomains meines Servers zugänglich, z. B. `git.domain.com` für GitLab und `registry.domain.com` für die Docker-Registry.

Da sowohl Palladio als auch PMX, die beiden Hauptanwendungen für die Performability-Analyse, verwendet wurden, sollten beide Anwendungen in Docker-Containern betrieben werden. Die zugehörigen Dockerfiles wurden von meinem Betreuer Sebastian Weber für

PMX bereitgestellt und von Thomas Weber für Palladio basierend auf seinem GitHub Repository adaptiert [3].

Die GitLab CI/CD-Pipeline wurde so konzipiert, dass sie in drei Stages unterteilt ist, wobei jede Stage ein eigenes Docker-Image verwendet. Diese Entkopplung der einzelnen Pipeline-Schritte ermöglicht eine einfache Wartung und Flexibilität bei zukünftigen Anpassungen.

Der Nutzer muss lediglich die Traces, die er für die Analyse und Simulation nutzen möchte, als Teil seines Repositorys bereitstellen. Sobald diese Traces in das Repository gepusht werden, wird die Pipeline automatisch angestoßen, und die Analyse sowie Simulation können erfolgen.

4 Verwendete Komponenten

4.1 Applikationssoftware

Die für die Analyse und Simulation eingesetzten Programme sind PMX und Palladio. PMX generiert aus Traces ein Performance Model, das von Palladio für Simulationen genutzt wird. Im Rahmen des Praktikums und für die CI/CD-Pipeline wurden beide Anwendungen als Blackbox betrachtet.

4.2 NixOS

Für die Serverumgebung habe ich mich für NixOS entschieden, da dieses Betriebssystem durch sein einzigartiges Paketverwaltungssystem eine präzise und reproduzierbare Konfiguration ermöglicht. NixOS erlaubt es, sämtliche Komponenten, wie Docker und GitLab, sauber in einer deklarativen Form zu verwalten und aufzusetzen, was den gesamten Prozess der Servereinrichtung erheblich vereinfacht hat.

In meiner `configuration.nix`-Datei habe ich die gesamte Umgebung definiert, einschließlich der Einrichtung von GitLab, GitLab Runner und der Docker-Registry.

4.3 Gitlab

Um GitLab auf meinem Server zu betreiben, habe ich die Konfiguration über NixOS vorgenommen. Der folgende Abschnitt aus meiner configuration.nix beschreibt die GitLab-Konfiguration:

```
gitlab = {
  enable = true;
  databasePasswordFile = "/var/keys/gitlab/db_password";
  initialRootPasswordFile = "/var/keys/gitlab/root_password";
  https = true;
  host = "git.dumustbereitsein.de";
  port = 443;
  user = "git";
  databaseUsername = "git";
```

```
  group = "git";
  smtp = {
    enable = true;
    address = "localhost";
    port = 25;
  };
  secrets = {
    dbFile = "/var/keys/gitlab/db";
    secretFile = "/var/keys/gitlab/secret";
    otpFile = "/var/keys/gitlab/otp";
    jwsFile = "/var/keys/gitlab/jws";
  };
  extraConfig = {
    gitlab = {
      email_from = "gitlab-no-reply@dumustbereitsein.de";
      email_display_name = "Vincenzos GitLab";
      email_reply_to = "gitlab-no-reply@dumustbereitsein.de";
    };
  };
};
```

In dieser Konfiguration habe ich GitLab so eingerichtet, dass es über die Subdomain git.dumustbereitsein.de per HTTPS erreichbar ist. Es wird eine lokale SMTP-Konfiguration verwendet, um Benachrichtigungen zu versenden, und die Zugangsdaten für die Datenbank sowie weitere sicherheitsrelevante Informationen werden in separaten Dateien gespeichert. HTTPS is notwendig, damit die Container-Registry mit GitLab kommunizieren kann. Die Secrets sind notwendig um GitLab überhaupt zu betreiben.

4.4 GitLab Runner

Der GitLab Runner spielt eine zentrale Rolle in der CI/CD-Pipeline. Er führt die einzelnen Jobs aus, die in den Pipeline-Stages definiert sind. Um eine CI/CD-Pipeline in GitLab umzusetzen, braucht man eine `.gitlab-ci.yml` Datei im Root-Verzeichnis des Repository. Der Runner benutzt diese dann, um die darin definierten Schritte auszuführen.

In meiner Konfiguration habe ich den GitLab Runner ebenfalls über NixOS eingerichtet:

```
    gitlab-runner = {
      enable = true;
      gracefulTermination = true;
    };
```

Der GitLab Runner ist so konfiguriert, dass er zuverlässig startet und sich bei Bedarf sicher beendet. Er wird mit dem GitLab-Server verbunden, um die CI/CD-Jobs auszuführen, die in den GitLab-Projekten definiert sind. Sobald ein Push in das Repository erfolgt, stößt

GitLab den Runner an, der dann die definierten Stages (wie Build, Test und Deployment) ausführt. Der Runner arbeitet dabei unabhängig mit isolierten Docker-Containern.

Der GitLab Runner muss manuell bei der GitLab-Instanz registriert werden, um die Verbindung herzustellen. Dies erfolgt über den Befehl `gitlab-runner register`, bei dem der Runner mit einer spezifischen URL und einem Registrierungstoken aus der GitLab-Instanz verknüpft wird. Dadurch wird sichergestellt, dass der Runner in der CI/CD-Pipeline verfügbar ist und die definierten Jobs ausführen kann. Prinzipiell besteht allerdings auch die Möglichkeit, diesen Runner ebenfalls via Nix als Teil der Gitlab Instanz aufzusetzen, ohne manuelle Schritte tätigen zu müssen.

4.5 Docker

In meiner CI/CD-Pipeline setze ich mehrere Docker-Images ein, speziell für das Projekts erstellt wurden. Die Containerisierung stellt sicher, dass die Anwendungen in einer isolierten Umgebung laufen, was sowohl die Wartbarkeit als auch die Skalierbarkeit verbessert.

PMX-Dockerfile Für das PMX-Image verwende ich das `eclipse-temurin:11`-Image, das eine optimierte Java-Laufzeitumgebung bietet. Der Inhalt des Repositorys wird in den Container kopiert, und das `entrypoint.sh`-Skript wird ausführbar gemacht, damit der Container beim Start korrekt arbeiten kann. Diese einfache Konfiguration ermöglicht die Ausführung der PMX-Analyse in einer konsistenten Umgebung und wurde erstellt von meinem Betreuer Sebastian Weber.

```
1 FROM        eclipse-temurin:11
2
3 COPY        .        .
4 RUN        chmod +x entrypoint.sh
```

Palladio-Dockerfile Das Palladio-Image basiert auf einem bestehenden Image von Thomas Weber [3], das ich angepasst habe. Ich habe unnötige Skripte entfernt und die Experimentdaten sowie das adaptierte Automatisierungsskript `RunExperimentAutomation.sh` in das Image kopiert.

```
1 FROM        thomasweber/palladioexperimentautomation
2
3 RUN        rm -f /usr/*.sh
4 COPY        ExperimentData/        /usr/ExperimentData/
5 COPY        RunExperimentAutomation.sh        /usr/RunExperimentAutomation.sh
6 RUN        chmod +x /usr/RunExperimentAutomation.sh
```

Gnuplot-Dockerfile Für die Visualisierung der Analyseergebnisse habe ich ein Image basierend auf Alpine Linux erstellt, das Gnuplot und die nötigen Schriftarten installiert. Das Skript `plot.sh`, das die Diagramme erzeugt, wird ebenfalls in den Container kopiert und ausführbar gemacht. Durch die Verwendung von Alpine als Basisimage bleibt das Image sehr schlank, was die Ausführung in der Pipeline beschleunigt.

```
FROM alpine:latest

RUN apk add --no-cache gnuplot fontconfig ttf-dejavu

COPY plot.sh .
RUN chmod +x plot.sh
```

Diese Docker-Images bilden die Grundlage der CI/CD-Pipeline und sorgen dafür, dass die Analyse und Simulation in isolierten, reproduzierbaren Umgebungen ausgeführt werden können.

5 Implementierung der CI/CD Pipeline

Die Software, die für die Performability-Analyse genutzt wird, ist bereits am Forschungszentrum Informatik (FZI) im Einsatz. Dort wird GitLab als zentrale Plattform für Versionskontrolle und CI/CD verwendet. Allerdings erfordert der Zugriff auf diese GitLab-Instanz spezielle Mitarbeiterrechte und eine VPN-Verbindung, die mir nicht zur Verfügung standen. Um dennoch mit ähnlichen Voraussetzungen arbeiten zu können, musste ich eine eigene Serverumgebung einrichten und die Gegebenheiten des FZI, insbesondere die GitLab-Umgebung, reproduzieren.

Ein wesentlicher Bestandteil dieser Reproduktion war das Einrichten einer eigenen Docker-Registry, wie sie auch am FZI genutzt wird. Das Aufsetzen der Docker-Registry erfolgte ohne Hilfestellung durch den Betreuer. Die GitLab CI/CD-Pipeline ist in drei Stages unterteilt: extract, simulate und plot. Jede Stage verwendet ein eigenes Docker-Image, das ich zuvor erstellt und in meine Docker-Registry hochgeladen habe. Die Konfiguration der Pipeline erfolgt in der `gitlab-ci.yaml`-Datei, die den Ablauf und die Abhängigkeiten der einzelnen Schritte definiert.

```
stages:
  - extract
  - simulate
  - plot
```

Extract-Stage In dieser Stage wird das PMX-Image verwendet. Das Skript in dieser Stage extrahiert die notwendigen Daten und speichert sie in einem Verzeichnis namens `results/`, das als Artefakt für die nächsten Schritte verfügbar gemacht wird. Das Docker-Image für diese Stage ist in der Registry hinterlegt:

```yaml
pmx:
  image: "registry.dumustbereitsein.de/vincenzo/pmx:latest"
  stage: extract
  script:
    - /entrypoint.sh
  artifacts:
    paths:
      - results/
```

Simulate-Stage Die Simulation erfolgt in dieser Stage mit dem Palladio-Image. Hier werden die extrahierten Dateien aus der vorherigen Stage verwendet, und IDs in den Modelldateien werden durch neue Werte ersetzt. Die alten IDs müssen überschrieben werden, da diese in `Generated.experiments` und `measuring2.monitorrepository` referenziert werden. `Generated.experiments` und `measuring2.monitorrepository` sind hard-coded im Docker Image. Wenn man diese IDs dort nicht überschreibt, Palladio sucht nach Dateien die nicht existieren. Die neuen IDs kommen aus den im ersten Schritt durch PMX generierten Dateien, die auf den Eingabetraces beruhen.

Der Simulator SimuLizar des Palladio Ansatzes wird mit den Modellen als Eingabe ausgeführt. Die Ergebnisse werden als Artefakte in einem Ordner `result/` gespeichert:

```yaml
palladio:
  image: "registry.dumustbereitsein.de/vincenzo/palladio_runtime:latest"
  stage: simulate
  dependencies:
    - pmx
  script:
    # Step 1: Copy extracted files from results/ to /usr/ExperimentData/model/
    - cp results/extracted.* /usr/ExperimentData/model/
    - cp results/measuring2.* /usr/ExperimentData/model/

    # Step 2: Extract the new IDs and replace them in Generated.experiments and
    ↪ other files
    - new_allocation_id=$(sed -n '2s/.*id="\([^"]*\)".*/\1/p' \
    /usr/ExperimentData/model/extracted.allocation)
    - new_monitor_id=$(sed -n '2s/.*id="\([^"]*\)".*/\1/p' \
    /usr/ExperimentData/model/measuring2.monitorrepository)

    # Update Generated.experiments
    - sed -i "15s/#.*/#$new_allocation_id\"\/>/" \
    /usr/ExperimentData/model/Experiments/Generated.experiments
    - sed -i "16s/#.*/#$new_monitor_id\"\/>/" \
    /usr/ExperimentData/model/Experiments/Generated.experiments
    # Update measuring2.monitorrepository
    - sed -i '12s/href="measuring\./href="measuring2\./' \
```

```
    /usr/ExperimentData/model/measuring2.monitorrepository
    - sed -i '19s/href="measuring\./href="measuring2\./' \
    /usr/ExperimentData/model/measuring2.monitorrepository

    # Step 3: Execute the container's main script
    - /usr/RunExperimentAutomation.sh

    - mv /result/ ./result/
  artifacts:
    paths:
      - result/
```

Plot-Stage In der letzten Stage wird das Gnuplot-Image verwendet, um die Ergebnisse der Simulation grafisch darzustellen. Das Skript generiert Diagramme, die als Artefakte im Verzeichnis `plots/` als .png gespeichert werden:

```
gnuplot:
  image: "registry.dumustbereitsein.de/vincenzo/gnuplot:latest"
  stage: plot
  dependencies:
    - palladio
  script:
    - /plot.sh
  artifacts:
    paths:
      - plots/*.png
```

Jede dieser Stages ist klar voneinander getrennt und nutzt die bereitgestellten Artefakte der vorhergehenden Stages. Dies sorgt für eine modulare und leicht wartbare CI/CD-Pipeline, die flexibel an unterschiedliche Anforderungen angepasst werden kann.

6 Fazit

Alle im Rahmen des Projekts gesetzten Ziele wurden erfolgreich erreicht. Am Ende der Pipeline werden die Ergebnisse der Simulation in Form von leicht verständlichen Grafiken bereitgestellt, die eine schnelle Interpretation der Simulationsergebnisse erlauben.

Es gibt jedoch noch Optimierungspotenzial, insbesondere im Umgang mit den bereitgestellten Traces. Die GitLab-CI-Pipeline könnte erweitert werden, um je nach Branch-Namen unterschiedliche Simulationen durchzuführen oder verschiedene Traces automatisch zu verwenden. Weiterhin könnte man bereits das Sammeln der Laufzeitdaten in die Pipeline integrieren, sodas die Simulationen immer den aktuellsten Stand des Projektes benutzen. Diese Erweiterungen könnten die Flexibilität der Pipeline weiter steigern.

References

[1] URL: https://se.informatik.uni-wuerzburg.de/software-engineering-group/tools/pmx/.

[2] URL: https://www.palladio-simulator.com/.

[3] URL: https://github.com/TomWerm/Palladio-Docker.