

**به نام خدا**

**گزارش 60 درصد پیشرفت پروژه کنترل کیفبت نرم‌افزار**

**نام سرباز : محمد خادم حسینیه**

**نام استاد راهنما : دکتر مرتضی جوان**

شهریور 1401

فهرست مطالب

[مقدمه 4](#_Toc112256769)

[فصل اول : بررسی شاخص‌های کنترل کیفیت نرم‌افزار 5](#_Toc112256770)

[1-1- شاخص‌های کیفیت نرم‌افزار 6](#_Toc112256771)

[1-1-1- قابلیت اطمینان 6](#_Toc112256772)

[1-1-2- امینت 8](#_Toc112256773)

[1-1-3- قابلیت نگهداری 9](#_Toc112256774)

[1-1-5- اندازه 13](#_Toc112256776)

[1-2- بررسی شاخص‌های نرم افزاری با استفاده از یادگیری ماشین 17](#_Toc112256779)

[1-2-1- بررسی ویژگی‌های شاخص‌های نرم افزاری 18](#_Toc112256780)

[فصل دوم: بررسی ناهنجاری در سیستم سوییفت با استفاده از لاگ‌های سیستم…………….. 22](#_Toc112256782)

[2-1- تشکیل گراف کنترل جریان برای لاگ‌ها 23](#_Toc112256783)

[2-1-1- انواع لاگ 23](#_Toc112256784)

[2-1-2- تشکیل پدر و فرزند برای هر لاگ 24](#_Toc112256785)

[2-1-3- مرج کردن گراف‌ها 25](#_Toc112256786)

[2-1-4- شناسایی ناهنجاری 26](#_Toc112256787)

[2-2- تشکیل گراف حالت برای لاگ‌ها 26](#_Toc112256788)

[2-3- بررسی انواع لاگ‌های سویفت 28](#_Toc112256789)

[2-3-1- رسم گراف حالت برای داده‌های سوییفت 32](#_Toc112256790)

[فصل سوم : مصورسازی………………….. 38](#_Toc112256791)

[3-1-1- پایگاه داده گراف 39](#_Toc112256792)

[3-1-2- استفاده از Neo4j 39](#_Toc112256793)

[3-1-6- معایب استفاده از پایگاه داده گراف: 41](#_Toc112256797)

[3-1-7- راه اندازی Neo4j: 41](#_Toc112256798)

[3-1-9- کدهای پیاده سازی کلاس گراف neo4j 46](#_Toc112256800)

[3-1-10- اضافه کردن programname، method و user-agent به عنوان ویژگی های نود در Neo4j 48](#_Toc112256801)

[فصل چهارم: سیستم کشف ناهنجاری.............................................................................................................................51](#_Toc112256802)

[4-1- معماری سوییفت و اهمیت کشف ناهنجاری 52](#_Toc112256803)

[4-1-1- الگوریتم کشف ناهنجاری 56](#_Toc112256804)

[4-1-2- بررسی چند نمونه لاگ ناهنجار 61](#_Toc112256805)

مقدمه

کنترل کیفیت یک پروژه نرم‌افزاری از مهم‌ترین ابعاد آن است. کنترل کیفیت نرم‌افزار می‌تواند مفاهیم و جنبه‌های متفاوتی داشته باشد. در این پروژه ما از دو جهت این مفهوم را بررسی می‌کنیم. جهت اول، بررسی کیفیت یک پروژه از نظر میزان کیفیت برنامه‌نویسی است. پارامترهای مختلفی در این زمینه تعریف می‌شود که به طور کامل در این مطالعه بررسی خواهند شد. این پارامترها، میزان مشابهت، وابستگی و شاخص‌های تست را زیر ذره‌بین قرار می‌دهند. شاخص تست به بررسی نتایج و میزان دربرگیرندگی تست‌های واحد می‌پردازد و به‌وسیله داده‌های به‌دست آمده از آن‌ها قضاوت می‌کند که نرم‌افزار تا چه اندازه مطابق با رویه‌های مورد انتظار که برای آن‌ها تست تهیه شده عمل می‌کند. هم‌چنین مشخص می‌کند چه مقدار از کل کدهای موجود شامل خط‌های کد و شرط‌ها تحت پوشش تست قرار گرفته‌اند. در ادامه این بخش به بررسی شاخص‌های نرم‌افزاری با استفاده از یادگیری ماشین می‌پردازیم. هدف این قسمت پیش‌بینی میزان تلاش لازم جهت انجام پروژه است. دانستن این امر بسیار حایز اهمیت است چرا که کم یا زیاد تخمین زدن زمان لازم برای یک پروژه باعث تخمین غلط یک شرکت از میزان وسعت و پیچیدگی آن پروژه می‌شود و میزان هزینه‌ای که برای این پروژه در نظر میگیرد را تحت تاثیر قرار می‌دهد.

در قسمت دوم این مطالعه، ما سعی می‌کنیم از لاگ‌های یک نرم‌افزار، ناهنجاری را تشخیص دهیم. برای این منظور ابتدا سعی می‌کنیم گراف سلامت سیستم را رسم کنیم و پس از آن درصدد کشف ناهنجاری در یک سیستم نرم‌افزاری برآییم. به‌طور خاص در این قسمت ما نرم‌افزار سوییفت را مورد بررسی قرار می‌دهیم و کیفیت الگوریتم‌های خود را می‌سنجیم.

1. فصل اول : بررسی شاخص‌های کنترل کیفیت نرم‌افزار
   1. شاخص‌های کیفیت نرم‌افزار

در این قسمت به بررسی شاخص‌های کیفیت نرم‌افزار می‌پردازیم. این شاخص‌ها شامل قابلیت اطمینان، امنیت، قابلیت نگه‌داری،تکرارها، اندازه، پیچیدگی و تست و پوشش هستند.

* + 1. قابلیت اطمینان

این شاخص بیان می‌کند که نرم‌افزار از لحاظ صحت عملکرد و پیروی از قوانین و الگوهای بهینه در چه جایگاهی قرار دارد. وجود باگ در نرم‌افزار، عدم پیروی از قوانین عمومی یا اختصاصی زبان، وجود آسیپ‌پذیری و مواردی از این دست می‌توانند قابلیت اطمینان نرم‌افزار را کاهش دهند.

این شاخص به‌وسیله استخراج و شمارش باگ‌ها و مشکلات و شدت آن‌ها سنجیده می‌شود.

برای مثال سیستم سونار[[1]](#footnote-1)با شمارش خطاهای استخراج شد و در نظر گرفتن شدت آن‌ها برچسبی با عنوان رتبه‌بندی اطمینان به کد اختصاص می‌دهد و درجه قابلیت اطمینان آن را منطبق با آن‌چه در پایین لیست شده، تعیین می‌کند:

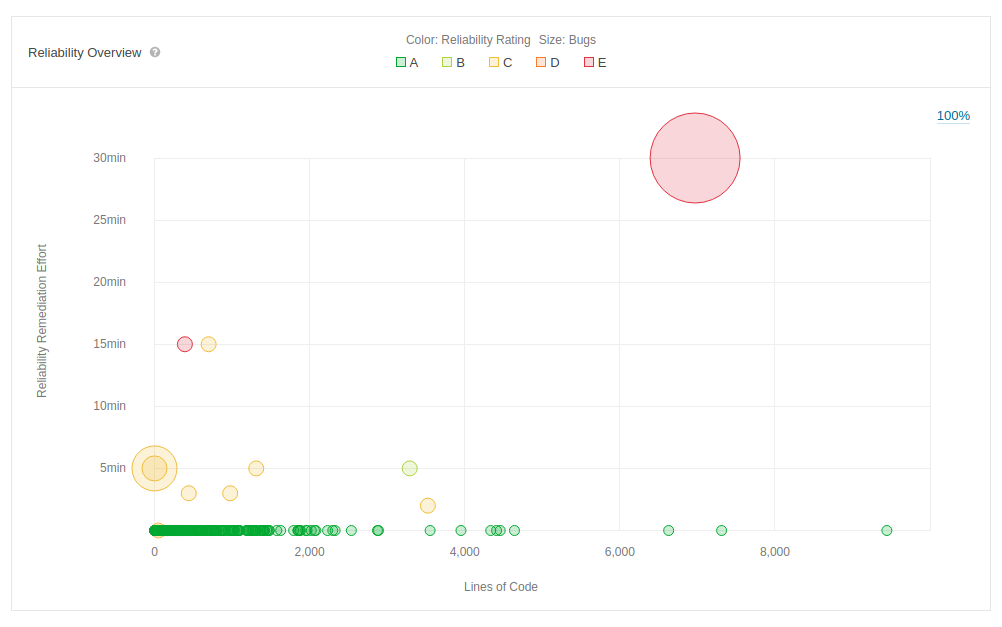
1. بدون خطا
2. حداقل یک مشکل فرعی
3. حداقل یک مشکل اصلی
4. حداقل یک مشکل بحرانی
5. حداقل یک مشکل مسدود کننده

علاوه‌بر این درجه‌بندی، شاخص هزینه اصلاح قابلیت اطمینان[[2]](#footnote-2) برای سورس کد ارائه می‌شود. این شاخص که در واحد زمان ارائه می‌شود بیان می‌کند چه‌ میزان زمان برای برطرف کردن همه‌ی خطاهای موجود در سیستم و دستیابی به بیشینه قابلیت اطمینان نیاز است. این زمان بر مبانی ساعت و روز بیان می‌شود که در آن هر ۱ روز معادل با ۸ ساعت در نظر گرفته می‌شود. نرم‌افزار سوییفت[[3]](#footnote-3) به دلیل وجود ۳۴ خطا که ۲ مورد از آن‌ها بلاکر هستند از این شاخص درجه اطمینان E دریافت کرده است. مابقی این موارد شامل ۳۱ خطای بحرانی و ۱ خطای فرعی هستند.



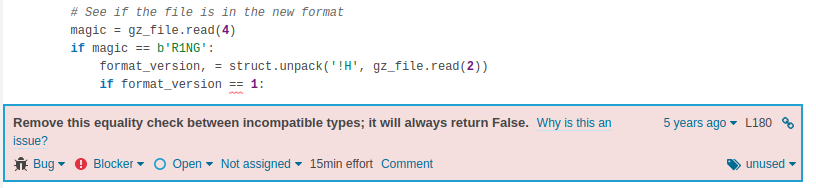
شکل 1 : بررسی سوییفت در سیستم sonar

هزینه اصلاح قابلیت اطمینان برای برای سورس کد سوئیفت ۱ ساعت و ۲۸ دقیقه محاسبه شده و نسبت آن به تعداد خط کد‌های هر بخش در جدول زیر ترسیم شده است.



شکل 2 : تعداد خط کدهای سوییفت

برای مثال در پایین یکی از خطاهای بلاکر تشخیص داده شده را مشاهده می‌کنیم.



شکل 3 : بلاکر تشخیص داده شده در سوییفت

* + 1. امینت

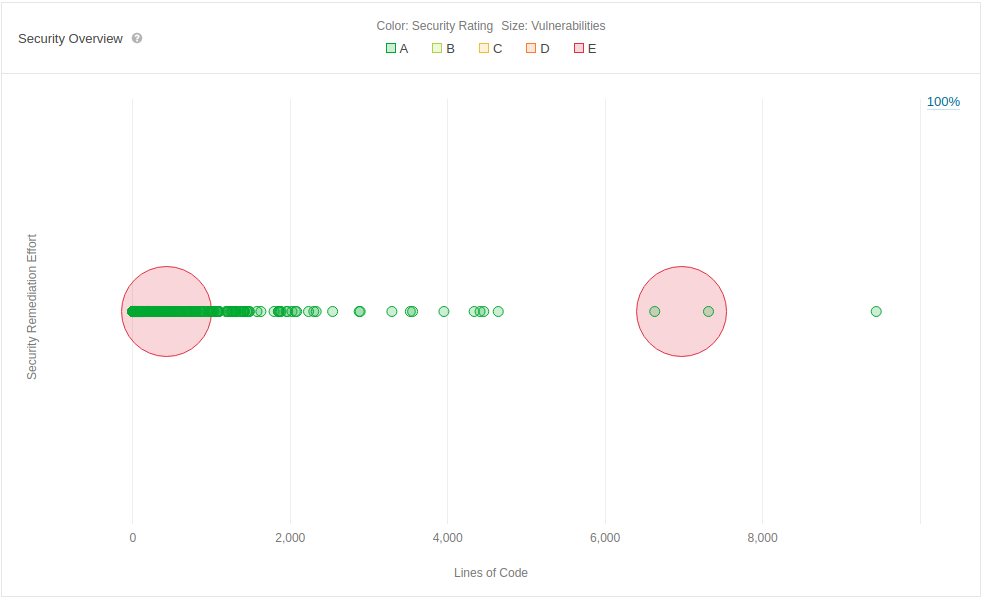
این شاخص بیان می‌کند که نرم‌افزار از نظر مقاومت در مواجهه با حملات در چه سطحی قرار دارد. نرم‌افزار ها در مقابل لیستی از آسیب‌پذیری‌های عمومی (مانند SQL Injection و XSS ها) و آسیب‌پذیری‌های اختصاصی یک زبان (مثل استفاده از backtick در پایتون) بررسی می‌شوند و وجود هر یک از این موارد در آن‌ها گزارش می‌شود. شاخص امنیت با بررسی تعداد و شدت نقاط آسیب‌پذیری استخراج شده مورد سنجش قرار می‌گیرد.

برای مثال سیستم سونار با شمارش این آسیب‌پذیری‌ها و در نظر گرفتن شدت آن‌ها برچسبی با عنوان رتبه‌بندی امنیت به کد اختصاص می‌دهد و درجه امینت آن را منطبق با آن‌چه در پایین لیست شده، تعیین می‌کند:

1. بدون آسیب‌پذیری
2. حداقل یک آسیب‌پذیری فرعی
3. حداقل یک آسیب‌پذیری اصلی
4. حداقل یک آسیب‌پذیری بحرانی
5. حداقل یک آسیب‌پذیری مسدود کننده

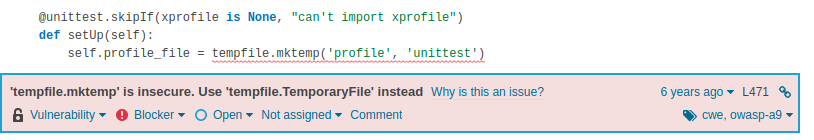
علاوه‌بر این درجه‌بندی، شاخص هزینه اصلاح امنیت[[4]](#footnote-4) برای سورس کد ارائه می‌شود. این شاخص که در واحد زمان ارائه می‌شود بیان می‌کند چه‌ میزان زمان برای برطرف کردن همه‌ی خطاهای موجود در سیستم و دستیابی به بیشینه قابلیت اطمینان نیاز است. این زمان بر مبانی ساعت و روز بیان می‌شود که در آن هر ۱ روز معادل با ۸ ساعت در نظر گرفته می‌شود. نرم‌افزار سوییفت به دلیل وجود ۲ آسیب‌پذیری امنیتی که هر ۲ مورد آن‌ها بلاکر هستند از این شاخص درجه اطمینان E دریافت کرده است.

هزینه اصلاح امنیت برای سورس کد سوئیفت صفر (با چشم پوشی از مقدار کوچک به‌دست آمده) محاسبه شده و نسبت آن به تعداد خط کد‌های هر بخش در جدول زیر ترسیم شده است.



شکل 4 : نسبت هزینه اصلاح امنیت به تعداد خط کد

برای مثال در پایین یکی از آسیب‌پذیری‌هایی که بلاکر تشخیص داده شده را مشاهده می‌کنیم.

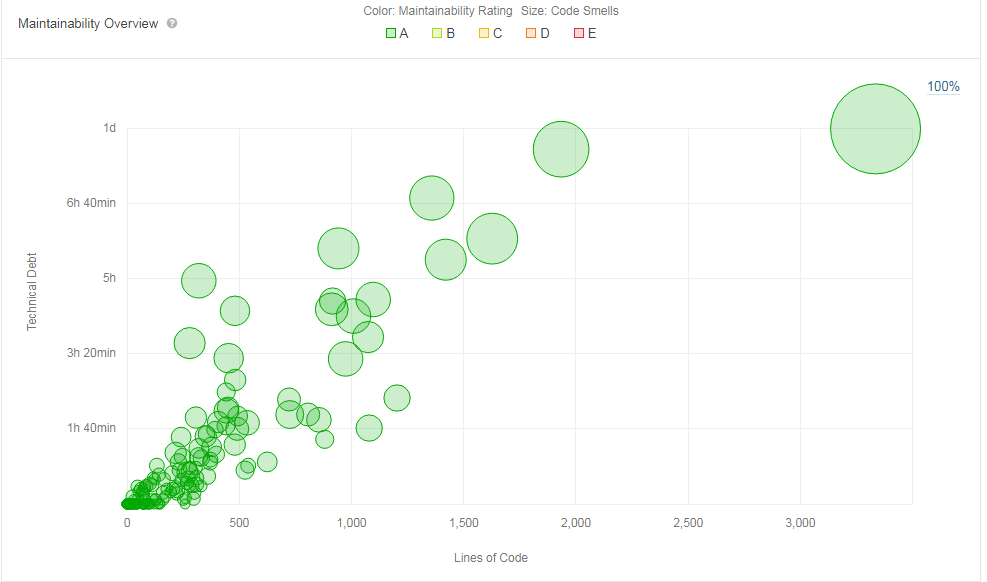


شکل 5: آسیب‌پذیری بلاکر

* + 1. قابلیت نگهداری

این شاخص بیان می‌کند که نگهداری نرم‌افزار تا چه اندازه دشوار و زمان‌بر است و مثلا برای برطرف کردن یک خطا به‌طور متوسط چه مقدار زمان باید صرف شود. کدهای مشکوک یکی از سنجه‌هایی است که برای اندازه‌گیری این شاخص استفاده می‌شوند. برای مثال وجود خط کد‌های طولانی، استفاده از چندین if و else که می‌تواند با یک switch case جایگزین شود از این موارد هستند.

در همین ارتباط مفهوم بدهی فنی[[5]](#footnote-5) مطرح می‌شود. که زمان کلی برای برطرف کردن همه‌ی کدهای مشکوک را شامل می‌شود. این زمان‌ها معمولا بسته به نوع مشکل به‌صورت ثابت‌هایی به دقیقه برای آن در نظر گرفته می‌شوند که مجموع آن‌ها بیان کننده بدهی فنی نرم‌افزار است. وضعیت کلی این شاخص در سوییفت بصورت زیر است.



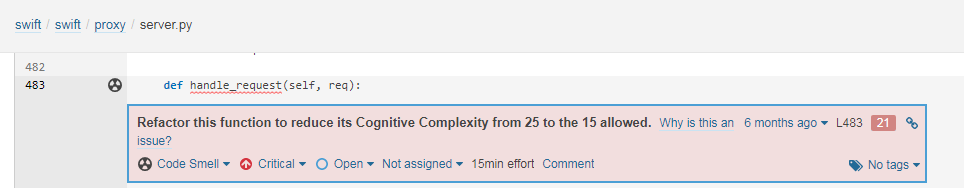
شکل 6 : شاخص بدهی فنی برای سوییفت

بدهی فنی در این پروژه حدود ۱۰۸ روز تخمین زده شده است که در بخش های مختلف بصورت زیر است:



شکل 7 : بدهی فنی در قسمت‌های مختلف سوییفت

برای مثال در پایین یکی از کدهای مشکوک که بحرانی تشخیص داده شده را مشاهده می‌کنیم.



شکل 8 : کدهای مشکوک

* + 1. تکرارها

این شاخص نشان‌دهنده میزان تکرار در سورس کد نرم‌افزار است. تکرار می‌تواند خط کد تکراری، بلاک کد تکراری یا فایل تکراری باشند. وجود تکرار در کد باعث افزایش اندازه کد، پیچیدگی نگهداری و افزایش زمان اعمال تغییرات می‌شود.

* خط کد تکراری: تعداد خط کدهایی که در یک نرم‌افزار عینا تکرار شده‌اند.
* فایل تکراری: تعداد فایل‌هایی که در یک نرم‌افزار عینا تکرار شده‌اند.

بلاک کد تکراری: حداقل n دستور پشت‌سرهم و تکراری بدون در نظر گرفتن تعداد خطوط و توکن‌ها. که مقدار n برای زبان‌های برنامه‌نویسی مختلف متفاوت است. پارامتر n در زبان‌هایی مثل جاوا و پایتون برابر ۱۰ و در زبانی مانند Cobol برابر با ۱۰۰ است. علاوه بر این شاخص تراکم تکرار [[6]](#footnote-6) بیان می‌کند چند درصد از کدها در سورس کد یا به ازای هر فایل تکراری هستند. در سورس کد سوئیفت تراکم تکرار کل پروژه برابر ۶/۱ درصد است. در زیر لیست برخی از فایل‌های دارای بیش‌ترین تراکم تکرار ارائه شده است.



شکل 9 : فایل‌های دارای بیشترین تراکم در سوییفت

همچنین در مجموع ۱۷۹۵۴ خط کد تکراری، ۱۳۸۶ بلاک کد تکراری و ۹۳ فایل تکراری در پروژه وجود دارد.



شکل 10 : نمونه ارور در سوییفت

* + 1. اندازه

شاخص اندازه بیان‌گر بزرگی نرم‌افزار است که تعداد دایرکتوری‌ها، فایل‌ها، کلاس‌ها، توابع، خط کد و دستورات است. اعدادی که از شمارش این موارد به‌دست می‌آیند می‌توانند بینشی در مورد بزرگی یک نرم‌افزار ارائه کنند.

برای شمارش خط کدها از معیار شمارش خط‌های کد و غیر کامنت استفاده می‌شود.

* خط[[7]](#footnote-7): همه خطوط موجود در یک فایل.
* خط کامنت [[8]](#footnote-8): همه‌ی خط‌های کامنت شده که حداقل یک کاراکتر غیر از space یا انواع tab یا کاراکترهای خاص مربوط به کامنت در آن وجود دارد.
* خط کد[[9]](#footnote-9): تعداد خط‌‌های از فایل که حداقل یک کاراکتر غیر از space یا انواع tab در آن وجود دارد و جزو کامنت‌های آن زبان برنامه‌نویسی محسوب نمی‌شود.

با استفاده از همین تعاریف مفهوم دیگری به نام تراکم کامنت نیز معرفی می‌شود که به شکل زیر محاسبه می‌گردد.

Density of comment lines = Comment lines / (Lines of code + Comment lines) \* 100

به‌وسیله این معادله برای مثال اگر تراکم کامنت یک فایل ۵۰ درصد باشد یعنی تعداد خط‌های کامنت با تعداد خط‌های کد برابر هستند و اگر ۱۰۰ درصد باشد به این معنی‌است که کل فایل را کامنت تشکیل می‌دهد.

آمار استخراج شده برای سورس کد سوئیفت به‌شرح زیر است:

Lines of Code: 222,345

Lines: 294,385

Functions: 13,148

Classes: 1,409

Files: 364

Comment Lines: 37,600

Comments (%): 14.5%

* + 1. پیچیدگی

شاخص پیچیدگی بیان‌گر میزان پیچیدگی یک کد از لحاظ تو در تو بودن روال‌ها و قابل فهم بود کد است. این شاخص با دو سنجه‌ی اصلی پیچیدگی چرخه‌ای و پیچیدگی مفهومی اندازه‌گیری می‌شود.

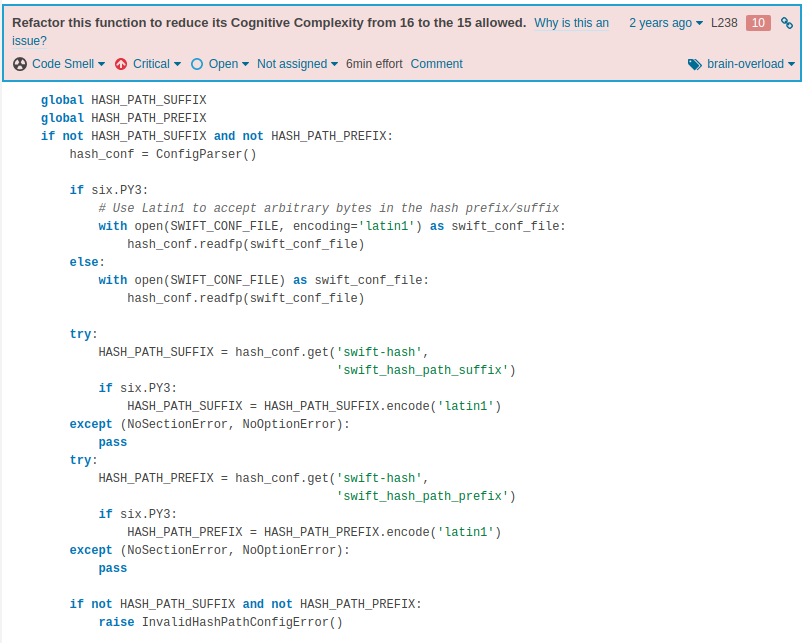
پیچیدگی چرخه‌ای[[10]](#footnote-10): این پیچیدگی بر مبنای تعداد مسیرها موجود در کد محاسبه می‌شود. هر بار که رویه کنترلی کد انشعاب پیدا می‌کند (برای مثال استفاده از بلاک‌های شرطی) این پیچیدگی یک واحد افزایش پیدا می‌کند. نحوه محاسبه این پیچیدگی با توجه به وجود کلمات کلیدی متنوع در هر زبان بسته به زبان‌های برنامه‌نویسی متفاوت است.

پیچیدگی مفهومی[[11]](#footnote-11): پیچیدگی مفهومی نشان می‌دهد که فهمیدن رویه کد چه مقدار دشوار است. برای محاسبه این فاکتور با سه قانون کلی در نظر گرفته می‌شود:

* در نظر نگرفتن ساختارهایی که اجازه می‌دهند چندین دستور در قالب یک دستور shorthand بازنویسی شوند.
* افزودن یک واحدی پیچیدگی برای هر شکستگی [[12]](#footnote-12) در ساختار خطی رویه‌ها.
* افزودن پیچیدگی زمانی که ساختارهای رویه‌شکن تو در تو هستند.

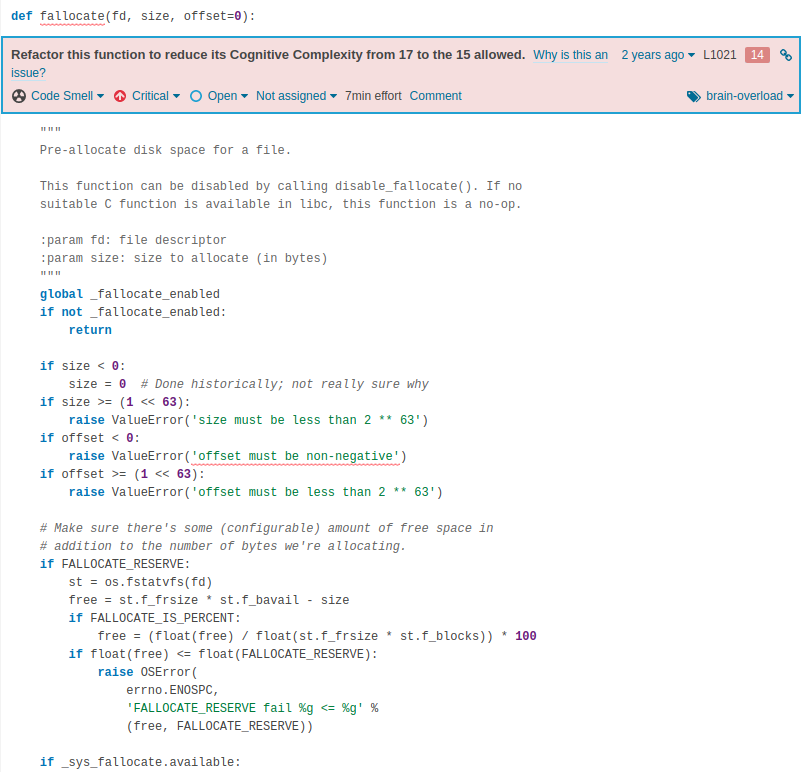


شکل 11 : پیچیدگی چرخه‌ای کد سوییفت

در سورس کد سوئیفت مجموع پیچیدگی‌ چرخه‌ای برابر با ۲۷۳۸۸ است که در تصویر زیر تعداد از پیچیده‌ترین فایل‌های این سورس کد از نظر پیچیدگی‌ چرخه‌ای مشاهده می‌شود: 

شکل 12 : نمونه‌ای از پیچیدگی چرخه‌ای

همین‌طور مجموع پیچیدگی‌ مفهومی در سورس کد سوئیت برابر با ۲۴۴۹۴ است که در تصویر زیر تعداد از پیچیده‌ترین فایل‌های این سورس کد از نظر پیچیدگی‌ مفهومی مشاهده می‌شود:



شکل 13 : پیچیدگی مفهومی سوییفت



شکل 14 : پیچیدگی مفهومی در قسمت‌های مختلف سوییفت

* + 1. تست و پوشش

این شاخص به بررسی نتایج و میزان دربرگیرندگی تست‌های واحد می‌پردازد و به‌وسیله داده‌های به‌دست آمده از آن‌ها قضاوت می‌کند که نرم‌افزار تا چه اندازه مطابق با رویه‌های مورد انتظار که برای آن‌ها تست تهیه شده عمل می‌کند. هم‌چنین مشخص می‌کند چه مقدار از کل کدهای موجود شامل خط‌های کد و شرط‌ها تحت پوشش تست قرار گرفته‌اند. از جمله سنجه‌های مرتبط با تست واحد می‌توان از تعداد تست‌های واحد، تعداد تست‌های دارای خطا، تعداد تست‌های شکست خورده با خطای پیش‌بینی نشده و تراکم موفقیت تست‌ها نام برد.

تعداد خط کد قابل پوشش، تعداد خط کدهای پوشش داده شده، تعداد شرط‌ بر اساس خط کد، تعداد شرط‌های پوشش داده شده بر اساس خط کد و پوشش کلی از جمله سنجه‌های مرتبط با پوشش هستند.

* 1. بررسی شاخص‌های نرم افزاری با استفاده از یادگیری ماشین

در این بخش به بررسی شاخص‌های نرم‌افزاری با استفاده از یادگیری ماشین می‌پردازیم. هدف این قسمت پیش‌بینی میزان تلاش لازم [[13]](#footnote-13) جهت انجام پروژه است. شکل زیر این مفهوم را نشان می‌دهد. دانستن این امر بسیار حایز اهمیت است چرا که کم یا زیاد تخمین زدن زمان لازم برای یک پروژه باعث تخمین غلط یک شرکت از میزان وسعت و پیچیدگی آن پروژه می‌شود و میزان هزینه‌ای که برای این پروژه در نظر میگیرد را تحت تاثیر قرار می‌دهد. در ابتدا کمی در مورد شاخص‌های نرم افزاری و ویژگی‌های آن‌ها بحث می‌کنیم.

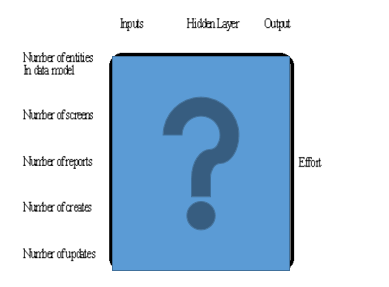


شکل 15 : سیتم تخمین‌زننده تلاش لازم برای یک پروژه

* + 1. بررسی ویژگی‌های شاخص‌های نرم افزاری

برای آن‌که سیستم پیشگوی یادگیری ماشین بتواند تخمین درستی از میزان تلاش لازم برای انجام یک پروژه داشته باشد باید ویژگی‌های ورودی به سیستم ویژگی‌های جامع، کامل و مستقل خطی باشند. شاخص‌های نرم‌افزاری از یکدیگر مستقل خطی نیستند و این خود موجب کاهش دقت سیستم می‌شود. یک راهی که می‌توان برای حل این مشکل پیشنهاد داد استفاده از روش آنالیز فاکتور[[14]](#footnote-14)است. مشکل دیگری که در شاخص‌های نرم‌افزاری به عنوان ورودی سیستم وجود دارد وجود دیتای گم‌شده[[15]](#footnote-15) در این شاخص‌هاست. مشکل دیگر وجود دامنه‌های مختلف در این شاخص‌هاست که خود فرآیند یادگیری را دچار مشکل می‌کند. از طرف دیگر، دیتابیس شامل این شاخص‌ها پر از دیتای نویزی[[16]](#footnote-16) هست. وجود این مشکلات حل مساله را دشوار می‌کند. به عبارت دیگر، ابتدا باید راهی برای حل این مسائل پیدا کرد و سپس مساله پیش‌بینی انجام شود.

در شکل زیر ویژگی‌های ورودی سیستم نمایش داده شده است. مساله یافتن ساختار مناسب به‌جای علامت سوال است.



شکل 16 : ورودی سیستم تخمین‌زننده تلاش نرم‌افزاری

* + 1. بررسی روش‌های تخمین زدن تلاش نرم‌افزاری

در این‌جا ما دو روش را برای تخمین زدن میزان تلاش بررسی می‌کنیم:

* رگرسیون
* شبکه‌های عصبی

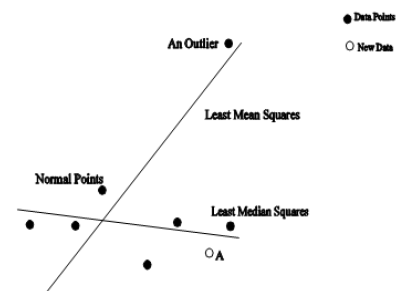
در مدل‌های آماری، **تحلیل رگرسیون** یک فرایند آماری برای تخمین روابط بین متغیرها می‌باشد. این روش شامل تکنیک‌های زیادی برای مدل‌سازی و تحلیل متغیرهای خاص و منحصر بفرد، با تمرکز بر رابطه بین متغیر وابسته و یک یا چند متغیر مستقل، می‌باشد. تحلیل رگرسیون خصوصاً کمک می‌کند در فهم اینکه چگونه مقدار متغیر وابسته با تغییر هرکدام از متغیرهای مستقل و با ثابت بودن دیگر متغیرهای مستقل تغییر می‌کند. بیشترین کاربرد تحلیل رگرسیون تخمین [امید ریاضی شرطی](https://fa.wikipedia.org/wiki/%D8%A7%D9%85%DB%8C%D8%AF_%D8%B1%DB%8C%D8%A7%D8%B6%DB%8C_%D8%B4%D8%B1%D8%B7%DB%8C) متغیر وابسته از متغیرهای مستقل معین است که معادل مقدار متوسط متغیر وابسته است وقتی که متغیرهای مستقل ثابت هستند. کمترین کاربرد آن تمرکز روی چندک یا پارامتر مکانی توزیع شرطی متغیر وابسته از [متغیر مستقل](https://fa.wikipedia.org/wiki/%D9%85%D8%AA%D8%BA%DB%8C%D8%B1_%D9%85%D8%B3%D8%AA%D9%82%D9%84) معین است. در همه موارد هدف تخمین یک تابع از متغیرهای مستقل است که **تابع رگرسیون** نامیده شده‌است. تحلیل رگرسیون به صورت گسترده برای پیش‌بینی استفاده شده‌است. تحلیل رگرسیون همچنین برای شناخت ارتباط میان [متغیر مستقل و وابسته](https://fa.wikipedia.org/wiki/%D9%85%D8%AA%D8%BA%DB%8C%D8%B1_%D9%85%D8%B3%D8%AA%D9%82%D9%84_%D9%88_%D9%88%D8%A7%D8%A8%D8%B3%D8%AA%D9%87) و شکل این روابط استفاده شده‌است. در شرایط خاصی این تحلیل برای استنتاج روابط عالی بین [متغیرهای مستقل و وابسته](https://fa.wikipedia.org/wiki/%D9%85%D8%AA%D8%BA%DB%8C%D8%B1%D9%87%D8%A7%DB%8C_%D9%85%D8%B3%D8%AA%D9%82%D9%84_%D9%88_%D9%88%D8%A7%D8%A8%D8%B3%D8%AA%D9%87) می‌تواند استفاده شود. هرچند این می‌تواند موجب روابط اشتباه یا باطل شود بنابراین احتیاط قابل توصیه است. تکنیک‌های زیادی برای انجام تحلیل رگرسیون توسعه داده شده‌است. روش‌های آشنا همچون [رگرسیون خطی](https://fa.wikipedia.org/wiki/%D8%B1%DA%AF%D8%B1%D8%B3%DB%8C%D9%88%D9%86_%D8%AE%D8%B7%DB%8C) و حداقل مربعات که پارامتری هستند، در واقع در آن تابع رگرسیون تحت یک تعداد محدودی از پارامترهای ناشناخته از داده‌ها تخمین زده شده‌است. رگرسیون غیر پارامتری به روش‌هایی اشاره می‌کند که به توابع رگرسیون اجازه می‌دهد تا در یک مجموعه مشخص از توابع با احتمال پارامترهای نامحدود قرار گیرند. رگرسیون با مینیمم کردن مجموع مجذورات خطا: این نوع از رگرسیون رایج‌ترین نوع رگرسیون در یادگیری ماشین است. رایج‌ بودن آن به این معنی نیست که عملکرد بهتری نسبت به بثیه انواع رگرسیون دارد بلکه دلیل رایج بودن آن راحتی استفاده از ان به دلیل وجود پکیج‌های اماده پایتون است. در این نوع رگرسیون هدف یافتن بردار w جهت مینیم کردن تابع زیر است:

(1)

که در ان مقادیر هدف هستند. این نوع از رگرسیون بسیار به دیتای نویزی[[17]](#footnote-17) حساس است. یعنی اگر دیتای پرتی در دیتابیس آموزش موجود باشد، سیستم سعی می‌کند خطی به دست بیاورد که این دیتای پرت را هم پوشش دهد. این امر باعث خطا در سایر دیتاها می‌شود. راه درست آن است که سیستم توجه کمی به دیتاهای پرت داشته باشد. از آن‌جایی که میانه داده‌ها به ندرت تحت تاثیر داده‌های پرت قرار می‌گیرد، روش زیر می‌تواند مشکل روش اول را حل کند. رگرسیون با مینیمم کردن میانه مجذورات خطا: در این نوع رگرسیون هدف یافتن مقادیر w جهت مینیمم کردن تابع زیر است:

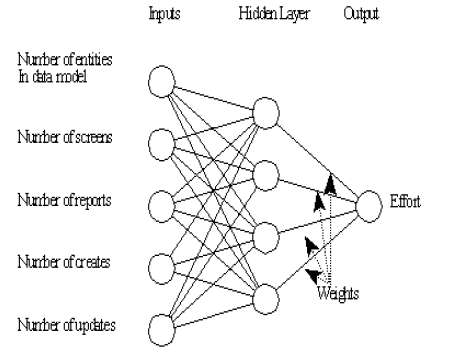
(2)

شکل زیر نشان می‌دهد که روش دوم تحت تاثیر داده‌های پرت قرار نمی‌گیرد.



شکل 16 : شرایطی که داده پرت ایجاد می‌کند.

شبکه‌های عصبی مصنوعی [[18]](#footnote-18)یا به زبان ساده‌تر شبکه‌های عصبی سیستم‌ها و روش‌های محاسباتی نوین برای [یادگیری ماشینی](https://fa.wikipedia.org/wiki/%DB%8C%D8%A7%D8%AF%DA%AF%DB%8C%D8%B1%DB%8C_%D9%85%D8%A7%D8%B4%DB%8C%D9%86%DB%8C)، [نمایش دانش](https://fa.wikipedia.org/wiki/%D9%86%D9%85%D8%A7%DB%8C%D8%B4_%D8%AF%D8%A7%D9%86%D8%B4) و در انتها اعمال دانش به دست آمده در جهت بیش‌بینی پاسخ‌های خروجی از [سامانه‌های پیچیده](https://fa.wikipedia.org/wiki/%D8%B3%D8%A7%D9%85%D8%A7%D9%86%D9%87%E2%80%8C%D9%87%D8%A7%DB%8C_%D9%BE%DB%8C%DA%86%DB%8C%D8%AF%D9%87) هستند. ایده‌ی اصلی این گونه شبکه‌ها تا حدودی الهام‌گرفته از شیوهی کارکرد [سیستم عصبی زیستی](https://fa.wikipedia.org/wiki/%D8%B3%D8%A7%D9%85%D8%A7%D9%86%D9%87_%D8%B9%D8%B5%D8%A8%DB%8C) برای [پردازش داده‌ها](https://fa.wikipedia.org/wiki/%D9%BE%D8%B1%D8%AF%D8%A7%D8%B2%D8%B4_%D8%AF%D8%A7%D8%AF%D9%87%E2%80%8C%D9%87%D8%A7) و [اطلاعات](https://fa.wikipedia.org/wiki/%D8%A7%D8%B7%D9%84%D8%A7%D8%B9%D8%A7%D8%AA) به منظور یادگیری و ایجاد [دانش](https://fa.wikipedia.org/wiki/%D8%AF%D8%A7%D9%86%D8%B4) می‌باشد. این سیستم از شمار زیادی عناصر پردازشی فوق‌العاده بهم‌پیوسته با نام [نورون](https://fa.wikipedia.org/wiki/%D9%86%D9%88%D8%B1%D9%88%D9%86) تشکیل شده که برای حل یک مسئله با هم هماهنگ عمل می‌کنند و توسط [سیناپس‌ها](https://fa.wikipedia.org/wiki/%D8%B3%DB%8C%D9%86%D8%A7%D9%BE%D8%B3_(%D8%B2%DB%8C%D8%B3%D8%AA%E2%80%8C%D8%B4%D9%86%D8%A7%D8%B3%DB%8C)) (ارتباطات الکترومغناطیسی) اطلاعات را منتقل می‌کنند. در این شبکه‌ها اگر یک سلول آسیب ببیند بقیه سلول‌ها می‌توانند نبود آن را جبران کرده، و نیز در بازسازی آن سهیم باشند. این شبکه‌ها قادر به یادگیری هستند. یادگیری در این سیستم‌ها به صورت تطبیقی صورت می‌گیرد، یعنی با استفاده از مثال‌ها وزن سیناپس‌ها به گونه‌ای تغییر می‌کند که در صورت دادن ورودی‌های جدید، سیستم پاسخ درستی تولید کند. به عنوان روش سوم، می‌توان از لایه آخر جهت تخمین زدن تلاش نرم‌افزاری استفاده کرد. شکل زیر این مساله را نشان می‌دهد. شبکه‌ی عصبی نشان داده شده در این شکل شامل یک لایه مخفی با 4 نورون است.



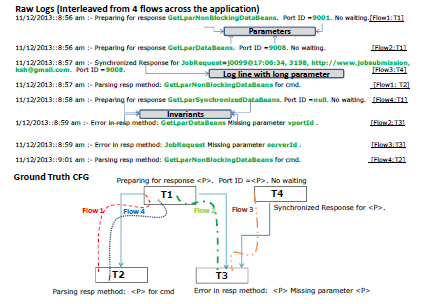
شکل 17 : شبکه عصبی تخمین تلاش نرم‌افزاری

1. فصل دوم: بررسی ناهنجاری در سیستم سوییفت با استفاده از لاگ‌های سیستم
   1. تشکیل گراف کنترل جریان[[19]](#footnote-19) برای لاگ‌ها

در این قسمت به بررسی نحوه تشکیل گراف کنترل جریان برای تشخیص ناهنجاری می‌پردازیم. برای این منظور ابتدا انواع و ساختار لاگ‌ها را بررسی می‌کنیم.

* + 1. انواع لاگ[[20]](#footnote-20)

در شکل زیر انواع و دسته‌بندی لاگ های یک سیستم نرم‌افزاری را مشاهده می‌کنید.



شکل 18 : ساختار لاگ‌ها

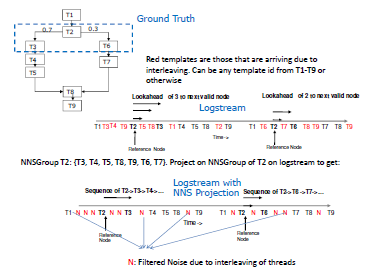
هر لاگ در سیستم نرم‌افزاری به دو دسته تقسیم می‌شود: قسمت تغییرپذیر[[21]](#footnote-21) و قسمت ثابت[[22]](#footnote-22). قسمت ثابت شامل تمپلیت‌های ثابتی مانند موارد زیر است:

* Prepared for response
* Synchronized Response
* Parsing resp method

اما قسمت متغیر شامل موارد مانند آدرس پورت[[23]](#footnote-23) و زمان انجام لاگ است. در شکل 1، مقصود از T1 و T2 و.. همان تمپلیت‌های ثابت هستند. در پایین این شکل، یک گراف کنترل جریان از لاگ‌ها ساخته شده است. در این شکل، Flow1 از تمپلیت‌های 1 و 2 استفاده میکند و همانطور هم که در شکل مشاهده می‌شود این فلو از T1 و T2 رد می‌شود. هدف در این قسمت ساختن چنین گرافی از لاگ‌ها به صورت کور است.

* + 1. تشکیل پدر [[24]](#footnote-24)و فرزند[[25]](#footnote-25) برای هر لاگ

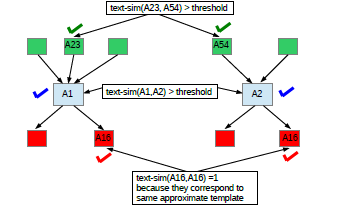
جهت آن که به یک گراف کنترل جریان برسیم باید برای هر لاگ تعدادی پدر و فرزند تشکیل شود. برای تشکیل پدر و فرزند برای هر تمپلیت[[26]](#footnote-26) باید توالی زمانی لاگ‌ها درنظر گرفته شود. به عنوان مثال فرض کنید که بخواهیم برای تمپلیت T2 پدر و فرزند استخراج کنیم. در شکل 2 این فرآیند نمایش داده شده است. دراین شکل، از یک روش آماری برای استخراج پدر و فرزندهای t2 استفاده شده است. برای آن‌که اساس این روش آماری توضیح داده شود ابتدا توالی لاگ‌های نمایش داده شده در شکل را درنظر بگیرید. تعدادی از این لاگ‌ها نویز هستند. با یک میانگین‌گیری ساده می‌توان متوجه شد که اکثر مواقع بعد از تمپلیت T1، تمپلیت T2 به وقوع می‌پیوندد. موارد نادر دیگری هم هستند که به عنوان نویز درنظر گرفته می‌شوند. با همین روش متوجه می‌شویم که بعد از T2 در 70 درصد موارد T3 و در 30 درصد موارد T6 به وقوع می‌پیوندد. موارد نادر دیگری هم هستند که به عنوان نویز درنظر گرفته می‌شوند. به همین سادگی پدر و فرزند تمپلیت T2 استخراج می‌شود. اما سوالی که یه وجود می‌آید این است: با چه حد آستانه‌ای[[27]](#footnote-27) موارد نویزی را تشخیص دهیم؟ برای پاسخ به این سوال از یک قاعده سرانگشتی استفاده می‌کنیم و آن هم این نکته است: اگر درصد حضور زیر 10 درصد باشد، آن مورد به عنوان نویز درنظر گرفته شود.



شکل 19 : تشکیل پدر و فرزند برای هر تمپلیت

* + 1. مرج کردن گراف‌ها

در شکل3 نحوه مرج کردن گراف‌های مربوط به هر تمپلیت نمایش داده شده است. در شکل زیر A1 و A2 دو تمپلیت هستند که همراه با پدرها و فرزندانشون نمایش داده شده ‌اند. اگر شباهت بین دو تمپلیت از نظر متنی[[28]](#footnote-28) بالای یک حد آستانه باشد دو گراف با همدیگر مرج می‌شوند و بین پدرها و فرزنداشنون اجتماع گرفته می‌شود. اما اگر این شباهت فقط بین فرزندان و یا پدرها باشد، دو گراف از محل آن شباهت بایکدیگر مرج می‌شوند این مرج شدن به همین صورت بین هر دو گراف تکرار می‌شود تا در نهایت به یک گراف کلی برسیم. گراف به دست آمده همان گراف کنترل جریان برای لاگ‌ها است. معیار شباهت هم، شباهت jaccard[[29]](#footnote-29) است.



شکل 20 : نحوه مرج کردن گراف‌ها

* + 1. شناسایی ناهنجاری[[30]](#footnote-30)

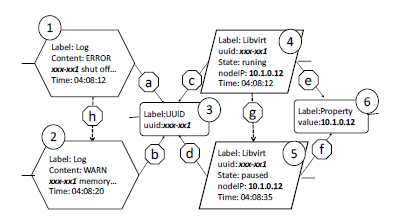
در یک سیستم نرم‌افزاری پس از تشکیل گراف کنترل جریان و گراف هر تمپلیت، دو حالت می‌تواند نشانه ناهنجاری باشد:

الف) در یک بازه زمانی مشخص، فرزندان یک تمپلیت مشاهده نشوند. اگر همه فرزندان مشاهده نشوند احتمال ناهنجاری بیشتر از حالتی است که یک یا تعدادی از فرزندان مشاهده نشوند.

ب) توزیع فرزندان یا پدران هر تمپلیت در یک بازه زمانی تغییر کند.

* 1. تشکیل گراف حالت[[31]](#footnote-31) برای لاگ‌ها

در این قسمت به بررسی نحوه ساخت گراف حالت برای لاگ‌های نرم افزاری می‌پردازیم. شکل 4 یک گراف حالت را نمایش می‌دهد.



شکل 21 : گراف حالت برای یک سیستم نرم افزاری

برای ساخت گراف حالت از لاگ‌های خام استفاده می‌شود. در ابتدای کار نیاز است که هر لاگ خام آنالیز شود و دو قسمت متغیر و اصلی از آن استخراج شود. پس از این مرحله، ساخت گراف حالت آغاز می‌شود. هر گراف حالت سه قسمت اصلی دارد: موجودیت[[32]](#footnote-32)، حالت[[33]](#footnote-33)، رخداد[[34]](#footnote-34). در شکل 4، هر مستطیل یک موجودیت، هر متوازی الاضلاع یک حالت و هر چندضلعی یک رخداد را نمایش می‌دهد. عضو دیگر نمودار شکل 4 یا‌ل‌ها هستند. یال‌ها به دو دسته فضایی[[35]](#footnote-35) و زمانی[[36]](#footnote-36) تقسیم می‌شوند. هر یال فضایی برای ایجاد یک رابطه بین یک موجودیت با یک رخداد و یا یک موجودیت با یک حالت استفاده می‌شود. هیچ‌گاه، یک رخداد با یک رخداد رابطه مستقیم ندارد و باید یک حالت و یا یک رخداد بین دو رخداد واسطه شود. حال به بررسی نحوه ساخت گراف فوق می‌پردازیم:

الف) حالت: این موارد در سیستم‌های نرم‌افزاری موارد مشخصی هستند و شامل مواردی مانند warning، error، run می‌شوند.

ب) رخداد: تشخیص موجودیت در یک سیستم نرم‌افزاری می‌تواند تاحدی چالشی باشد. معمولا به عنوان یک قاعده سرانگشتی موارد با این ویژگی به عنوان یک موجودیت درنظر گرفته می‌شوند: ویژگی‌هایی که موارد متغیر بسیاری دارند و این موارد متغیر در چند حالت و یا رخداد تکرار شده‌اند. برای مثال در شکل 4، 10.1.12 در دو حالت و xxx-xx1 در دو حالت و دو رخداد تکرار شده است. بنابراین این موارد به عنوان یک موجودیت درنظر گرفته می‌شوند.

پ) لبه فضایی[[37]](#footnote-37): یک لبه فضایی، یک حالت یا رخداد را به یک موجودیت متصل می‌کند اگر آن حالت یا رخداد شامل آن موجودیت باشد. این نکته در شکل 4 کاملا مشخص است.

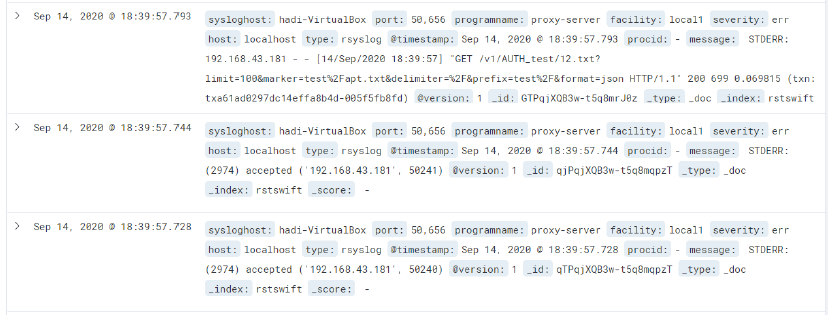
ت) لبه زمانی[[38]](#footnote-38) : برای رسم لبه زمانی ابتدا از نظر هر موجودیت، حالت ها و رخداد ها دسته‌بندی می‌شوند. به عبارت دیگر، حالت هایی که با رخداد مورد نظر در ارتباط هستند در یک گروه قرار می‌گیرند. همچنین، حالت هایی هم که با رخداد موردنظر در ارتباط هستند هم در یک دسته قرار می‌گیرند. سپس براساس توالی زمان اتفاق افتادن حالت ها و یا رخداد ها، بین هر دو رخداد یا هر دو حالت یک لبه زمانی رسم می‌شود. جهت این لبه زمانی براساس زمان است (از اتفاق زودتر به اتفاق دیرتر). این مورد در شکل 4 کاملا مشخص است.

* 1. بررسی انواع لاگ‌های سویفت

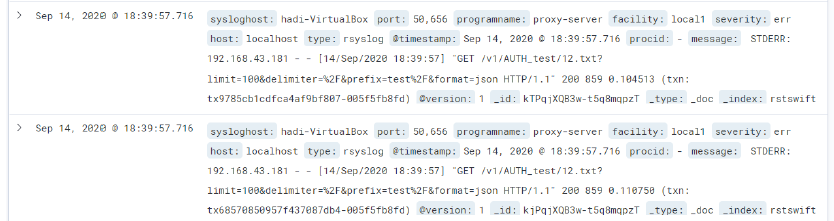
با توجه به آنکه هدف سیستم رسم گراف سلامت سیستم یا گراف جریان در سیستم است، ابتدا یک مطالع کامل برروی انواع لاگ های موجود در سوییفت برای سناریو آپلود و راه‌اندازی سرور انجام می‌دهیم. در این موارد سرورهای زیردخیل هستند:

* Object server
* Container server
* Proxy server
* Account server

در ابتدا سناریو آپلود را با هم بررسی می‌کنیم. فایل apt.txt را در سوییفت آپلود می‌کنیم. لاگ مربوط به پروکسی سرور به صورت زیر است :



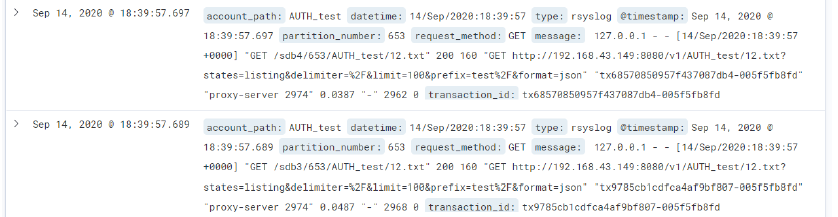
شکل 22 : نمونه اول لاگ پروکسی سرور در زمان آپلود

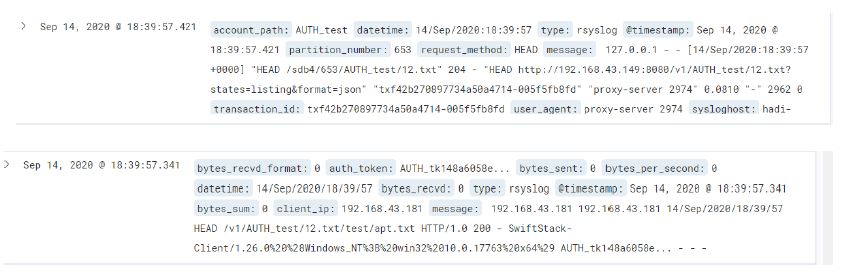


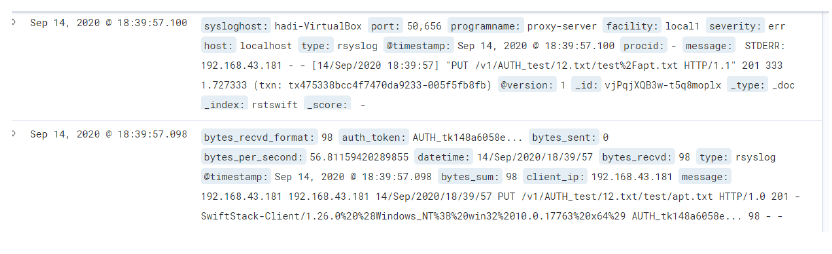
شکل 23 : نمونه دوم لاگ پروکسی سرور در زمان آپلود

لاگ مربوط به کانتینر سرور به صورت زیر است:



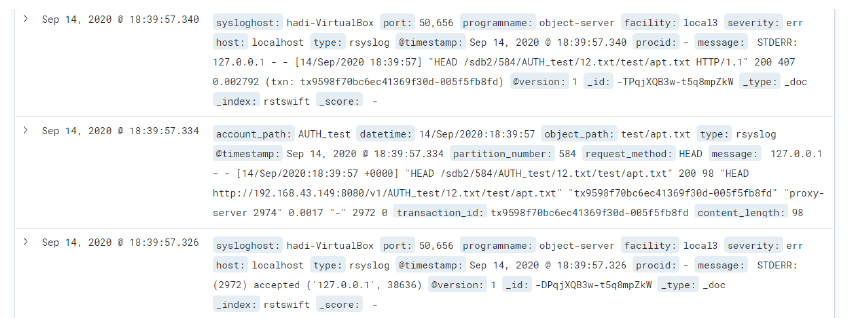






شکل 24 : لاگ مربوط به کانتینر سرور در هنگام آپلود

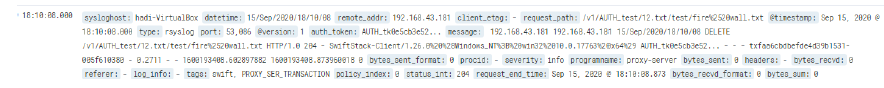
لاگ مربوط به آبجت سرور به صورت زیر است:



شکل 25 : لاگ آبجکت سرور در هنگام آپلود

برای سناریو حذف فایل از سوییفت لاگ سه سرور فوق به صورت زیر است:

پروکسی سرور:



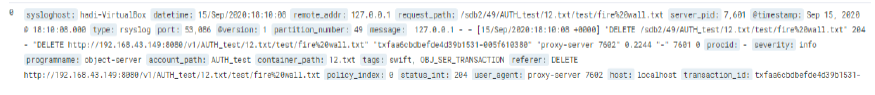
شکل 26 : لاگ پروکسی سرور در هنگام حذف

کانتینر سرور:



شکل 27 : لاگ کانتینر سرور در هنگام حذف

آبجکت سرور:

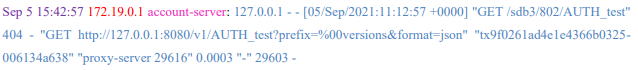


شکل 28 : لاگ آبجکت سرور در هنگام حذف

هر لاگ به صورت کلی در سامانه سوییفت به صورت زیر در syslog ثبت می‌شود :

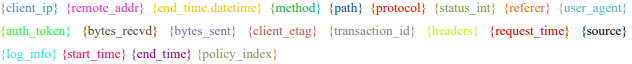
Datetime} {host} {prognamname} {message}}

که به ترتیب از سمت چپ نشان دهنده: تاریخ و زمان ثبت لاگ، آدرس IP سرویس تولیدکننده لاگ ، نام سرویس تولیدکننده لاگ و متن لاگ هستند.



**شکل 29 : نمونه لاگ پروکسی سرور**

همانگونه که ملاحظه میفرمایید، متن لاگ برای سرویسهای پروکسی و نودهای ذخیرهسازی ساختاریافته میباشد که در ادامه به تشریح ساختار آن پرداخته شده است؛ اما برای سایر سرویسها، متن لاگ از فرمت مشخصی تبعیت نمی‌کند.لاگ‌های پروکسی تمام درخواستهای API خارجی ارسال شده به سرور پروکسی را شامل میشوند. توضیح فیلدهای این ساختار لاگ در جدول1 آورده شده است. ساختار لاگ پروکسی سرور به صورت زیر است :



**شکل 30 : نمونه فرمت لاگ پروکسی**

جدول یک – بررسی ساختار لاگ

|  |  |
| --- | --- |
| فیلد | مقدار |
| Client ip | آدرس IP کلاینت نهایی با توجه به هدر درخواست |
| Remote addr | آدرس IP سرویس خارجی |
| End\_time.datetime | زمان و تاریخ درخواست |
| method | نوع عملیات درخواستی |
| path | مسیر درخواست |
| protocol | پروتکل انتقال (HTTP یا HTTPS) |
| Status\_int | کد پاسخ |
| Referrer | درخواستی که به واسطه دریافت آن، فرستنده اقدام به ارسال درخواست فعلی کرده است. |
| User\_agent | نام سرویس طرف دیگر درخواست |
| Auth\_token | توکن احراز هویت |
| Bytes\_recvd | تعداد بایتهای خوانده شده از کلاینت به واسطه این درخواست |
| Bytes\_sent | تعداد بایتهای فرستاده شده به کلاینت در بدنه پاسخ |
| Client\_etag | مقدار Etag داده شده توسط کلاینت |
| Transaction\_id | شناسه یکتای تراکنش که مقدار آن برای همه درخواست‌های مربوط به یک تراکنش یکسان است |
| headers | هدرهای داده شده در درخواست |
| Request time | مدت زمان درخواست |
| source | مبنع درخواست |
| Log\_info | اطاعات اضافی |
| Start\_time | زمان شروع درخواست با رزلوشن بالا |
| End\_time | زمان خاتمه درخواست با رزلوشن بالا |
| Policy\_index | مقدار اندیس سیاست ذخیرهسازی |

* + 1. رسم گراف حالت برای داده‌های سوییفت

داده های سوییفت حالات مختلفی را در خود دارند. حالات مختلفی که می‌تواند برای این داده ها اتفاق بیفتد delete ، get، auth و put است. برای آن که گراف حالت سوییفت را به دست بیاوریم ابتدا دیتاست کاملی از این داده‌ها و موارد و حالات ذکرشده در سوییفت را تولید ‌می‌کنیم. دیتاست تولیشده در نهایت شامل 10000 لاگ از حالات ذکرشده سوییفت است. بررسی یک نمونه از لاگ های سوییفت :

container-server: 127.0.0.1 - - [08/Dec/2020:09:23:03 +0000] "PUT /sdb1/66/AUTH\_test/admin/10.xlsx" 201 - "PUT http://localhost:8091/sdb1/113/AUTH\_test/admin/10.xlsx" "txcd52f845a2834526b4c4d-005fcf45f7" "object-server 75" 0.0007 "-" 76 0  
Dec 8 09:23:03

ساختار لاگهای سوویفت ساختار مشخصی است. در ساختار این لاگ ها به طور معمول دو سرور پیامی را با هم رد و بدل میکنند. سرور های محدود هستند و شامل container server و proxy server و account server و object server هستند. پیام رد بدل شده بین این سرورها هم موارد put و delete و get و auth است. با توجه به ساختار مشخص این لاگ ها میتوان الگوریتم مطرح شده در مقاله را به در قالب مراحل زیر خلاصه سازی کرد:

**الف) مرحله خواندن فایل شامل لاگ ها**

f = open(fileName)

json\_read = json.load(f)

در این مرحله فایل خوانده می‌شود و به یک دیکشنری تبدیل می‌شود.

**ب) مرحله تبدیل لاگ ها به فرمت استاندارد**

در این قسمت هر لاگ به اجزای اصلی آن تبدیل می‌شود (نام سرورها و متودها). در این قسمت هر سرور به صورت زیر خلاصه میشود:

Proxy-server: p

Account-server: a

Object-server: o

Container server: c

یک نمونه از خلاصه سازی کد در این قسمت انجام میشود با مثال زیر قابل توضیح است :

نمونه لاگ :

container-server: 127.0.0.1 - - [08/Dec/2020:09:23:03 +0000] "PUT /sdb1/66/AUTH\_test/admin/10.xlsx" 201 - "PUT http://localhost:8091/sdb1/113/AUTH\_test/admin/10.xlsx" "txcd52f845a2834526b4c4d-005fcf45f7" "object-server 75" 0.0007 "-" 76 0  
Dec 8 09:23:03

خلاصه شده : o Auth C

def purifylogs(self, json\_read):

        pure\_logs = []

        transaction\_ids = []

        datetime = []

        for i in range(len(json\_read)):

            if "HEAD" in json\_read[i]["request\_method"]:

                try:

                    str1 = json\_read[i]["user\_agent"]

                    str2 = json\_read[i]["programname"]

                    pure\_logs.append(self.define(str1) + "-" + "HEAD" + "-" + self.define(str2))

                    transaction\_ids.append(json\_read[i]['transaction\_id'])

                    datetime.append(json\_read[i]['@timestamp'])

                    continue

                except:

                    pass

            if "GET" in json\_read[i]["request\_method"]:

                try:

                    str1 = json\_read[i]["user\_agent"]

                    str2 = json\_read[i]["programname"]

                    pure\_logs.append(self.define(str1) + "-" + "GET" + "-" + self.define(str2))

                    transaction\_ids.append(json\_read[i]['transaction\_id'])

                    datetime.append(json\_read[i]['@timestamp'])

                    continue

                except:

                    pass

            if "DELETE" in json\_read[i]["request\_method"]:

                try:

                    str1 = json\_read[i]["user\_agent"]

                    str2 = json\_read[i]["programname"]

                    pure\_logs.append(self.define(str1) + "-" + "DELETE" + "-" + self.define(str2))

                    transaction\_ids.append(json\_read[i]['transaction\_id'])

                    datetime.append(json\_read[i]['@timestamp'])

                    continue

                except:

                    pass

            if "PUT" in json\_read[i]["request\_method"]:

                try:

                    str1 = json\_read[i]["user\_agent"]

                    str2 = json\_read[i]["programname"]

                    pure\_logs.append(self.define(str1) + "-" + "PUT" + "-" + self.define(str2))

                    transaction\_ids.append(json\_read[i]['transaction\_id'])

                    datetime.append(json\_read[i]['@timestamp'])

                    continue

                except:

                    pass

        return np.array(pure\_logs), np.array(transaction\_ids), np.array(datetime)

    @staticmethod

    def define(string):

        if "proxy" in string:

            return "p"

        if "account" in string:

            return "a"

        if "container" in string:

            return "c"

        if "object" in string:

            return "o"

        if "python" in string:

            return "s"

**پ) مرحله دسته بندی اولیه با درنظر گرفتن transaction id**

در این مرحله، لاگهایی که آیدی یکسان دارند به عنوان یک فرآیند شناخته میشوند. در حقیقت ما لاگ های با آیدی یکسان را با حفظ ترتیب به عنوان یک فرآیند درنظر میگیریم.

    @staticmethod

    def extractFlows(pure\_logs, transaction\_ids, datetime):

        final\_logs = []

        final\_date\_time = []

        final\_tx = []

        unique\_transaction\_ids = np.unique(transaction\_ids)

        for i in range(len(unique\_transaction\_ids)):

            temp\_logs = []

            temp\_datetime = []

            temp\_tx = []

            print(i)

            for j in range(len(transaction\_ids)):

                if transaction\_ids[j] == unique\_transaction\_ids[i]:

                    temp\_logs.append(pure\_logs[j])

                    temp\_datetime.append(datetime[j])

                    temp\_tx.append(transaction\_ids[j])

            final\_logs.append(temp\_logs)

            final\_date\_time.append(temp\_datetime)

            final\_tx.append(temp\_tx)

        return final\_logs, np.array(final\_date\_time), np.array(final\_tx)

**ت) سورت کردن لاگ های هر فرآیند**

در این قسمت لاگ های هر فرایند با توجه به فیلد datetime سورت می‌شوند. این قسمت از آن نظر حائز اهمیت است که در فلو نهایی ترتیب درستی را نمایش دهیم.

@staticmethod

    def sortlogs(logs\_flows, datetime):

        for i in range(len(logs\_flows)):

            temp\_logs\_flows = copy.deepcopy(logs\_flows[i])

            temp\_date\_time = copy.deepcopy(datetime[i])

            for k in range(len(temp\_date\_time)):

                try:

                    temp\_date\_time[k] = int((temp\_date\_time[k].split("T")[1].split(":")[0])) \* 3600 + (

                        int(temp\_date\_time[k].split("T")[1].split(":")[1])) \* 60 + \

                                        float(temp\_date\_time[k].split("T")[1].split(":")[2].split("Z")[0])

                    # int((temp\_date\_time[k].split("2022:")[1].split(":")[0])) \* 3600 + (

                    # int(temp\_date\_time[k].split("2022:")[1].split(":")[1])) \* 60 + \

                    # int(temp\_date\_time[k].split("2022:")[1].split(":")[2])

                except:

                    temp\_date\_time[k] = 0

            sorted\_temp\_date\_time = np.array([y for y, x in sorted(zip(temp\_date\_time, temp\_logs\_flows))])

            sorted\_temp\_logs\_flows = np.array([x for y, x in sorted(zip(temp\_date\_time, temp\_logs\_flows))])

            logs\_flows[i] = sorted\_temp\_logs\_flows

        return logs\_flows

**ث) مرحله کلاستر کردن فرآیندها**

در این مرحله فرآیندهای یکسان با هم دیگر در یک کلاستر قرار میگیرند. در حقیقت یکی از آن ها به عنوان نماینده در ادامه الگوریتم حضور خواهدیافت. معیارهای کلاسترکردن فرآیندها به صورت است:

* برابر بودن طول فرآیند
* برابر بودن نظریر به نظیر لاگ ها

     def clusterlogs(self, sorted\_log\_flows, tx):

        his = [sorted\_log\_flows[0]]

        his\_tx = [tx[0]]

        for i in range(len(sorted\_log\_flows)):

            found = 0

            for j in range(len(his)):

                if len(his[j]) == len(sorted\_log\_flows[i]):

                    count = 0

                    for k in range(len(his[j])):

                        if his[j][k] == sorted\_log\_flows[i][k]:

                            count = count + 1

                            continue

                    if count == len(his[j]):

                        found = 1

            if found == 0 and len(sorted\_log\_flows[i]) > 1:

                his.append(sorted\_log\_flows[i])

                his\_tx.append(tx[i])

        return his, his\_tx

**ج) مرحله تولید فایل اکسل برای هر کلاستر**

در این مرحله، سعی می‌شود که برای هر کلاستر یک فایل اکسل تولید شود که نام آن فایل اکسل شماره کلاستر یا فلو به اضافه transaction\_id آن فلو است.

    def generate\_excel(self, clustered\_sorted\_log\_flows, tx):

        for i in range(len(clustered\_sorted\_log\_flows)):

            df = pd.DataFrame(clustered\_sorted\_log\_flows[i])

            filename = str(i) + "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" + str(tx[i][0]) + '.csv'

            df.to\_csv(filename)

1. فصل سوم : مصورسازی
   * 1. پایگاه داده گراف

گراف در واقع مجموعه ای موجودیت ها و ارتباط بین آن هاست، که به صورت منظم ذخیره سازی شده اند. معمولاً در پایگاه داده هایی که به صورت سطر و ستون ذخیره می شوند، سرعت دسترسی به اطلاعات خوب و منطقی به نظر میرسد. به خصوص زمانی که از Indexing استفاده شود، این در حالی است که در بسیاری از مواقع به دلیل نیاز به سرعت بالای دسترسی به اطلاعات، سرعت واکشی و درج داده ها، در پایگاه داده های رابطه ای، مناسب به نظر نمی رسد. این اتفاق معمولاً زمانی رخ می دهد که میخواهید، یک یا چندJoin مختلف، بر روی جداول مختلف یک پایگاه داده بزنید.

در پایگاه داده های رابطه ای بستگی به نوع سناریو که داریم بعضی اوقات نیاز هست که برای پیاده سازی روابط خیلی پیچیده ، داده ها را در قالب جداول متعدد ذخیره کنیم . این جا ما مجبور به نوشتن Query ها و Join های زیادی هستیم که بتوانیم خروجی و گزارش دلخواه مان را بسازیم . همینطور که در جریان هستید هرچقدر کوئری های ما بزرگ و پیچیده بشود و روابط بین جداول زیاد شود، هزینه اجرای دستور ما (Cost of Query) بالا میرود و سرعت اجرای دستورات کم‌تر می‌شود.

* + 1. استفاده از Neo4j

Neo4j یکی از معروف‌ترین سیستم مدیریت دیتابیس است که از خانواده دیتابیس‌های NoSQL به شمار می‌رود. Neo4j با Mysql و یا MongoDB متفاوت و دارای ویژگی‌های خاص خود است که این ویژگی‌ها، آن را در مقایسه با سایر سیستم‌های مدیریت دیتابیس، خاص می‌کند.

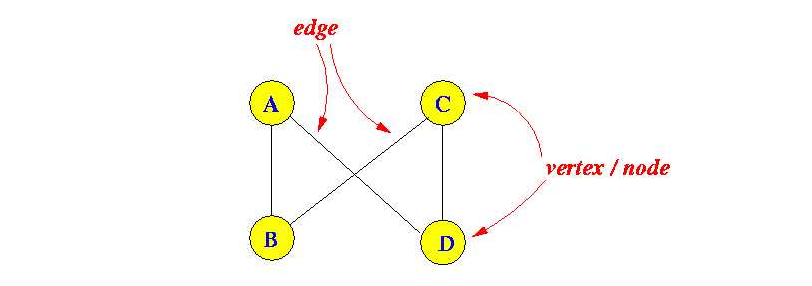
دیتابیس Neo4j داده‌ها را به صورت نمودار ذخیره و ارائه می‌کند و نه به‌صورت جدول یا JSON‌ .در دیتابیس Neo4j کل داده‌ها توسط nodeها نمایش داده می‌شوند و شما می‌توانید رابطه‌ای بین nodeها ایجاد کنید. این بدان معناست که کل مجموعه دیتابیس مانند یک نمودار خواهد بود، به همین دلیل آن را از دیگر سیستم‌های دیگر منحصر به فرد می‌کند.

برخی از ویژگی‌های منحصربه‌فرد وجود دارد که باعث می‌شود Neo4j را به عنوان جایگزینی برای دیگر سیستم‌های مدیریتی دیتابیس در نظر گرفت. همچنین باید گفت، دیتابیس Neo4j برای ایجاد روابط بین داده‌ها، نیازی به Primary key و Foreign key ندارد. در اینجا می‌توان هر ارتباطی بین هر node مورد نظر خود اضافه کنید. این قابلیت در Neo4j، آن را برای داده‌های شبکه (Network) بسیار مناسب می‌کند.



شکل 31 : گراف Neo4j

* + 1. مفاهیم اصلی Graph Database ها :
* گره‌ها یا node نشان دهنده موجودیت یا entitiy های ما هستند .. مثل مردم، کسب و کار ، حساب‌های کاربری.
* یال‌ها یا edge به‌طور کلی یال‌ها وظیفه نمایش روابط را در دیتابیس بر عهده دارند. همچنین بسیاری از اطلاعات مهم در یال‌ها ذخیره می‌شوند.



شکل 32 : رابطه گرافی

* + 1. چه موقع از ساختار گراف باید استفاده کرد ؟
* مواقعی که برنامه شما دارای داده ها با ساختار سلسله مراتبی (hierarchy) است. مانند چارت سازمانی.
* زمانی موجودیت های شما دارای روابط پیچیده چند به چند باشد.
  + 1. مزیت استفاده از پایگاه داده گراف:
* برای داده های متصل بسیار مناسب هست.
* برای انسان قابل خواندن و یادگیری و آسان هست.
* داده های نیم ساخت یافته را به راحتی ارایه می کند.
* برای بازیابی و پیمایش داده های متصل ساده هست.
  + 1. معایب استفاده از پایگاه داده گراف:
* در تعداد استفاده از گره ها و روابط و خواص دارای محدودیت هست.
* از sharding پشتیبانی نمی کند.



شکل 33 : نمونه ای از گراف neo4j

* + 1. راه اندازی Neo4j:

در این پروژه، برای استفاده از این ابزار، از نسخه داکر Neo4j استفاده شده که به سهولت و بدون توجه به سیستم عامل و سایر مشخصات محیط توسعه و تنها با داشتن سرویس داکر قابل استفاده است.

داکر برنامه‌ای رایانه‌ای و ابزاری است که ایجاد، توسعه و اجرای [اپلیکیشن‌ها](https://fa.wikipedia.org/wiki/%D8%A7%D9%BE%D9%84%DB%8C%DA%A9%DB%8C%D8%B4%D9%86) را با استفاده از کانتینر (container) آسان می‌کند. کانتینر به توسعه‌دهندگان این امکان را می‌دهد که تمام پیش‌نیازها و نیازمندی‌های اپلیکیشن خود را برای استفاده و اجرا جمع‌آوری کنند؛ مانند کتابخانه‌ها (Libraries) و زیرساخت‌های لازم.

می‌توان گفت که داکر به توسعه‌دهندگان این اطمینان را می‌دهد که می‌توانند اپلیکیشن خود را بدون نگرانی از سیستم‌های میزبان برنامه خود در مرحله توسعه و تست و بر روی سیستم‌های مختلف، بدون هیچ‌گونه اشکال و به صورت کاملاً مشابه اجرا کنند و مشکلی از بابت تغییر سیستم‌عامل‌های اجراکننده برنامه خود نخواهند داشت.

داکر تا حدی شبیه به ماشین مجازی (Virtual Machine) است با این تفاوت که در ماشین مجازی، قسمتی از سخت‌افزار سیستم به ماشین مجازی اختصاص داده می‌شود و روی آن یک سیستم‌عامل کامل نظیر ویندوز یا لینوکس نصب می‌شود. در واقع می‌توان گفت در ماشین مجازی امکانات سخت‌افزاری سیستم تقسیم می‌شود و بر روی هر قسمت، سیستم‌عامل بخصوصی بالا می‌آید اما در داکر این طور نیست. در داکر امکانات سخت‌افزاری به تناسب نیاز هر کانتینر به صورت موقت اختصاص داده می‌شود و داکر این امکان را فراهم می‌آورد که اپلیکیشن‌ها برای مثال روی [کرنل](https://fa.wikipedia.org/wiki/%DA%A9%D8%B1%D9%86%D9%84) لینوکس اجرا شوند. در این حالت دیگر نیازی به نصب پیش‌نیازها و نیازمندی‌هایی که اپلیکیشن ما می‌خواهد و به طور پیش‌فرض روی سیستم وجود ندارد، نیست.

کانتینر، یک واحد استاندارد نرم‌افزار است که کدها و تمام نیازمندی‌های آن را جهت اجرا و توسعه اپلیکیشن‌ها، تجمیع می‌کند و باعث می‌شود که این اجرا به طور سریع و قابل اعتماد انجام شود. کانتینرها، نرم‌افزار را از محیط خودش ایزوله می‌کنند و اطمینان می‌دهند که علی‌رغم تفاوت‌های احتمالی در سیستم‌های میزبان، نرم‌افزار به طرز مشابه و یکسان اجرا خواهد شد.

کانتینرهای داکر قسمتی از نرم‌افزار را در یک سیستم فایل کامل تعبیه می‌کند. به صورتی که شامل هر آنچه جهت اجرا شدن (مانند کد زمان اجرا، ابزارهای سیستم و تنظیمات سیستم) لازم است و هر آنچه که می‌تواند بر روی یک سرور نصب شود می‌باشد. این امر اجرای برنامه را به صورت ثابت در هر نوع محیطی تضمین می‌کند.

به طور کلی می‌توان گفت داکر این قابلیت را به ما می‌دهد تا یک اپلیکیشن را در یک محیط ایزوله به نام کانتینر (Container) اجرا کنیم. ایزوله بودن و امنیت کانتینرها باعث می‌شود که بتوانیم تعداد زیادی کانتینر را روی ماشین میزبان اجرا کنیم. کانتینرها سبک و سریع هستند زیرا نیازی ندارند که مانند ماشین‌های مجازی سربار Hypervisor را به ماشین تحمیل کنند و مستقیما روی هسته (Kernel) کامپیوتر سرویس‌دهنده اجرا می‌شوند. این بدان معناست که می‌توان تعداد بیشتری کانتینر را روی یک ماشین نسبت به حالتی که از ماشین مجازی استفاده می‌کنیم اجرا کرد. البته حتی این قابلیت را نیز داریم که از داکر درون ماشین مجازی نیز استفاده کنیم.

Container یک نمونه قابل اجرای یک Image است. می‌توان یک کانتینر را به وسیله Docker API یا کلاینت ایجاد، اجرا، حذف و یا متوقف کرد.

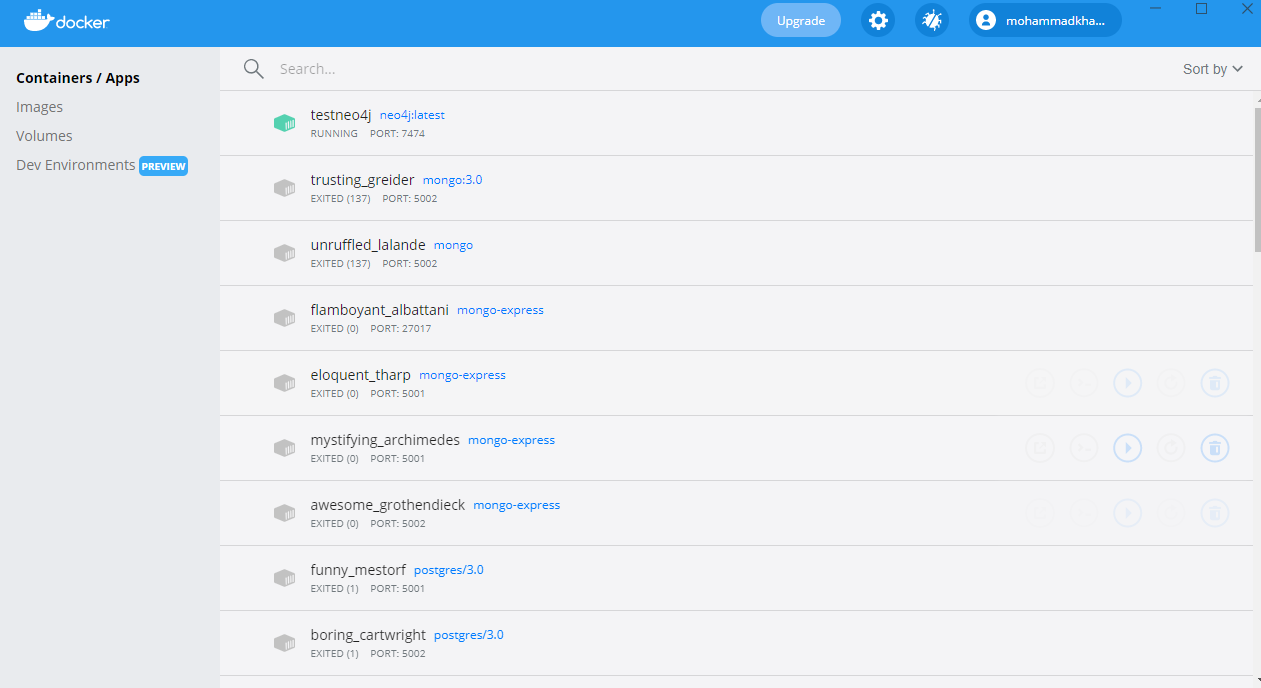
معمولا کانتینرها از دیگر کانتینرهای در حال اجرا به خوبی ایزوله می‌شوند. شما می‌توانید میزان ایزوله بودن کانتینرها را در مواردی مانند دسترسی شبکه‌ای، حافظه مورد استفاده آن‌ها و سیستم‌های پایه‌ای مورد استفاده‌شان تنظیم کنید.

یک کانتینر در واقع با Image آن و همه تنظیماتی که در زمان اجرای آن انجام می‌دهید تعریف می‌شود. هنگامی که کانتینر حذف می‌شود، هر تغییر فایلی که در آن به وجود آمده از بین میرود.

Docker Container Image یک پکیج نرم‌افزاری سبک، خوداتکا (Stand-alone) و قابل اجراست. ایمیج ها حاوی تمامی نیازمندی‌های اپلیکیشن برای اجرا مانند کدها، ابزارهای سیستمی، کتابخانه‌ها (Libraries) و تنظیمات سیستمی هستند. Imageها در زمان اجرا تبدیل به کانتینر شده و روی Docker engine اجرا می‌شوند. ایمیج ها برای هر دو نوعِ اپلیکیشن‌های بر پایه لینوکس و ویندوز در دسترس هستند.

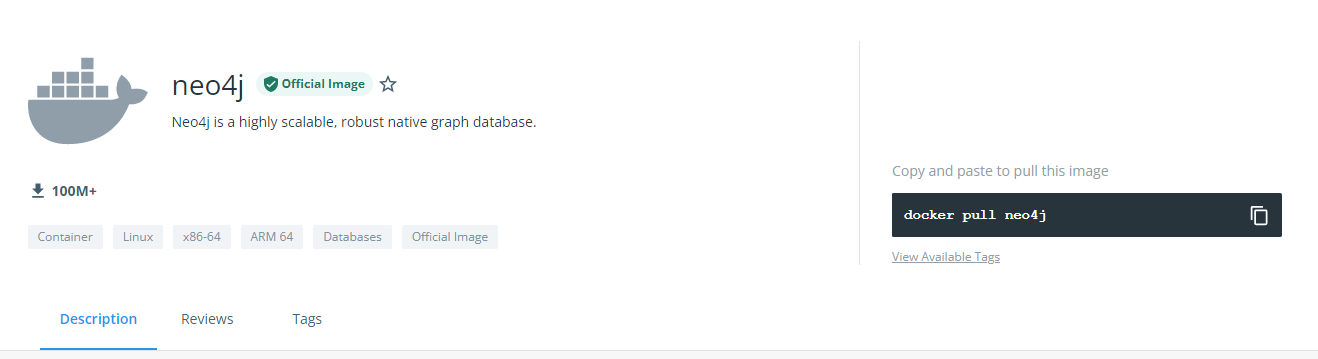
یک Image در واقع یک Read-Only Template حاوی دستوراتی برای ساخت یک کانتینر داکر است. معمولا استفاده ما از Image ها به این‌صورت است که با تغییر یکImage، Image دیگری را که برای ما مناسب‌تر است می‌سازیم. مثلاً می‌توانیم Image ای بسازیم که برپایه Image اوبونتو است اما یک سرور Apache نیز روی آن نصب می‌شود. شما میتوانید از Image های آماده استفاده کنید و یا Image مناسب خود را بسازید. برای ساختن Image جدید کافی است یک Dockerfile ساده بسازید که در آن دستوراتی برای نشان‌دادن چگونگی ساختن و اجرا کردن Image موجود است. وقتی که Dockerfile خود را تغییر می‌دهیم و Image جدیدی می‌سازیم، تنها بخش‌های جدید دوباره ساخته می‌شوند.

برای نصب داکر در ویندوز، بر روی بسته نرم افزاری Docker Desktop Installer.exe که قبلا دانلود کرده اید دو بار کلیک کنید تا نصاب اجرا شود. در پنجره آغازین فرایند نصب مطمئن باشید که تیک گزینه Enable Hyper-V Windows Features خورده باشد. سپس با زدن دکمه OK مجوز نصب داکر را صادر کرده و فرایند نصب را تا انتها مطابق دستورالعمل های خواسته شده توسط نصاب، کامل کنید. بعد از نصب موفق و مشاهده پیغام Installation Succeeded، دکمه Close and Logout را انتخاب کنید تا فرایند نصب کامل شود. با انتخاب دکمه مذکور، شما به طور خودکار از حساب کاربری ویندوزتان خارج خواهید شد لذا لازم است مجددا وارد حساب کاربری ویندوزتان شوید.



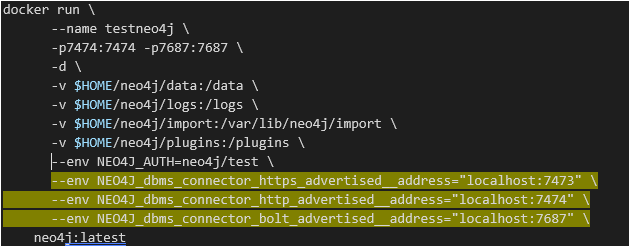
شکل 34 : تصویری از کانتینرها در داکر دسکتاپ

ایمیج Neo4j با تنظیمات پیش فرض‌ عامی که معمولا نیازی به تغییر ندارند در دسترس است[[39]](#footnote-39).



شکل 35 : ابزار neo4j در داکرهاب

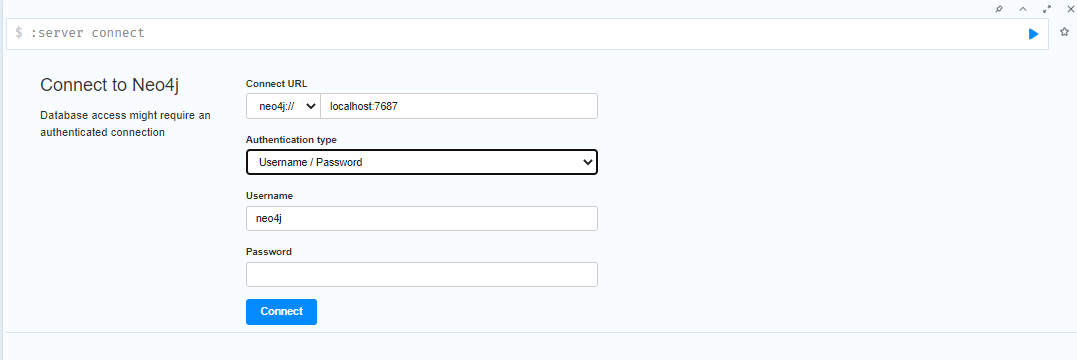
برای اجرای آن، پس از پول کردن ایمیج از داکرهاب، کافیست دستور زیر را از ترمینال اجرا کنیم:



شکل 36 : دستورات راه اندازی neo4j

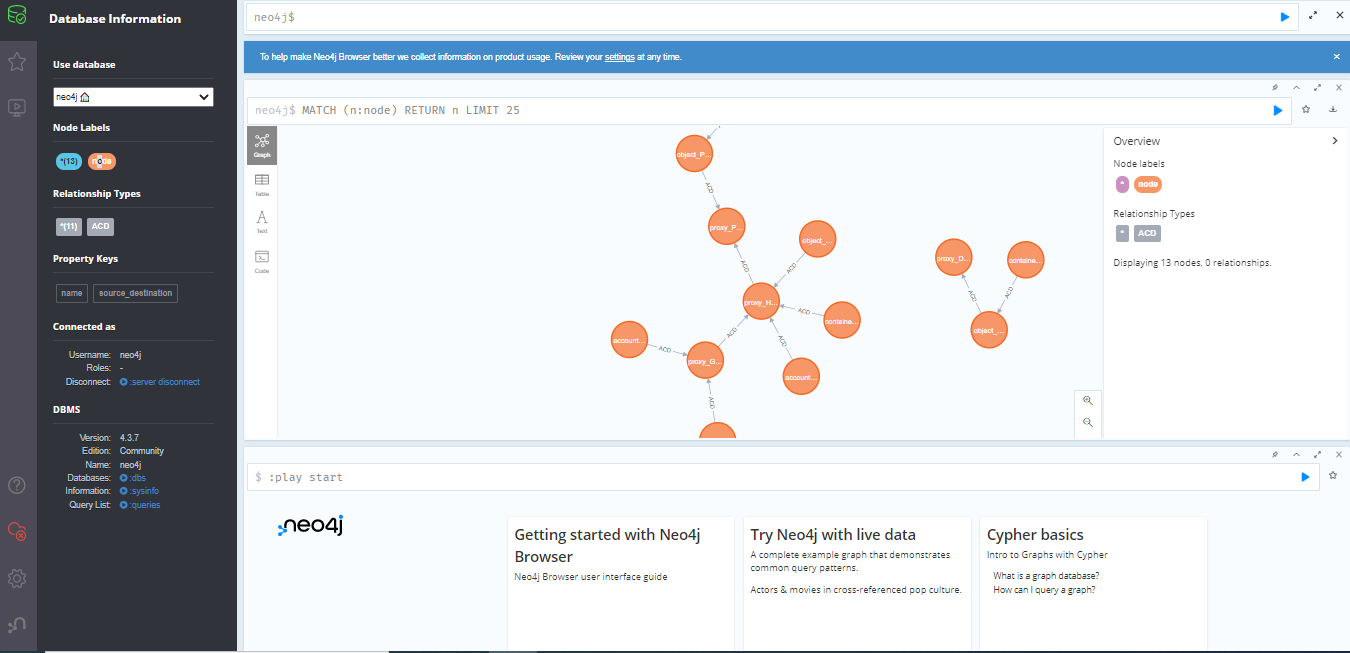
پس از راه اندازی موفق کانتینر، در اپلیکیشن پایتونی خود، با استفاده از کتابخانه neo4j و پورت 7678 و پروتکل Bolt می‌توانیم با این ابزار کار کرده و پیام‌های مورد نظر خود را که شامل دستورات مربوط به ایجاد و پاک کردن گره‌ها و یال‌ها می‌شود، برای آن ارسال کنیم.

همچنین با استفاده از مرورگر و روی پورت 7474 می‌توان به داشبورد Neo4j دسترسی داشت که به صورت بصری اطلاعات را نمایش می‌دهد، و همچنین می‌توان با استفاده از اسکریپت‌های به زبان cypher کوئری‌های مشابه SQL را در کنسول آن اجرا نمود و فیلترهای دلخواه را روی نودها و گره‌ها اعمال کرد.



شکل 37 : ورود به داشبرد neo4j

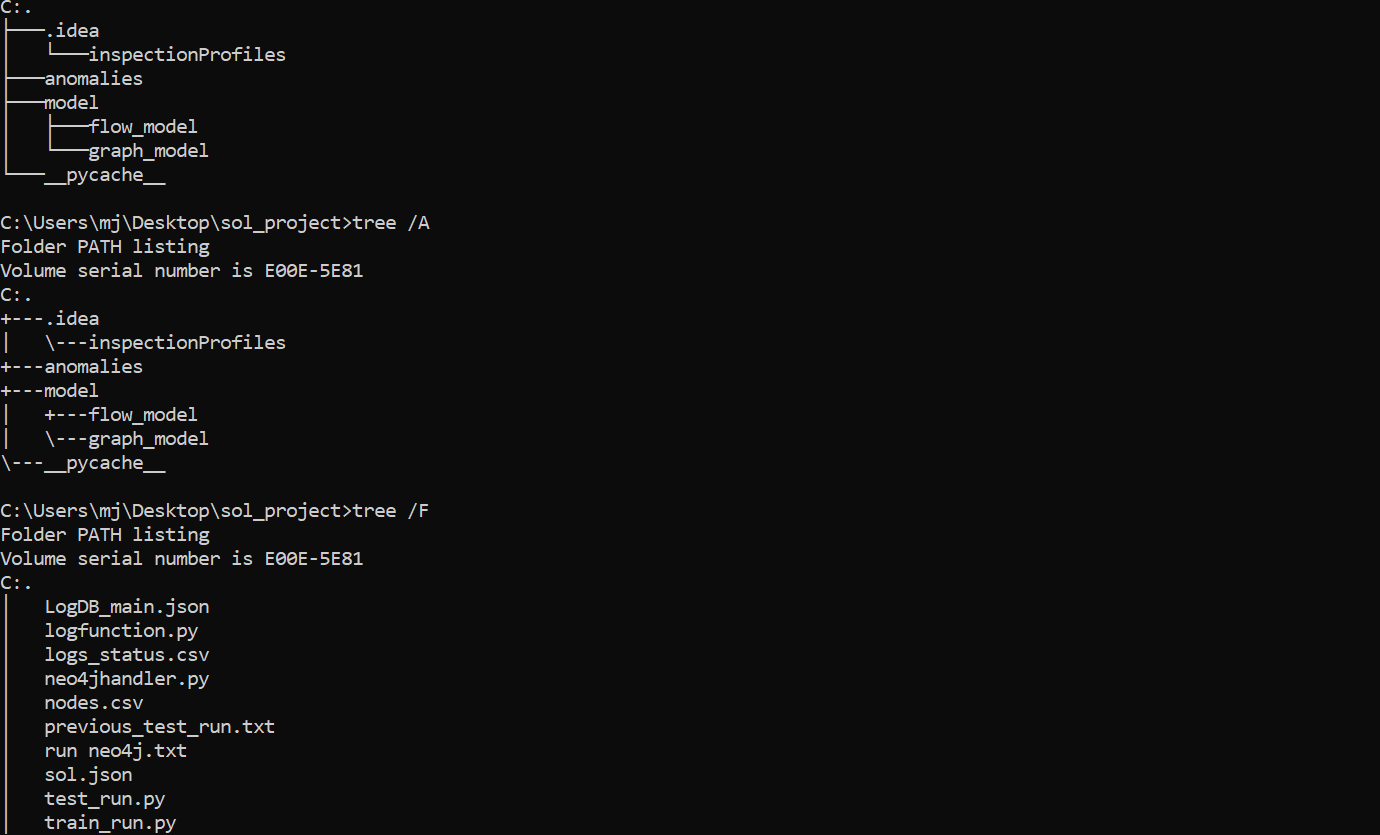
پس از وارد کردن یوزرنیم و پسورد وارد فضای زیر می‌شویم :



شکل 38 : داشبرد ne4j

* + 1. نکات مهم در استفاده از ابزار ne4j
* در استفاده از این ابزار حتما در ابتدا متودی با عنوان clear graph درست کنید و قبل از رسم هر گونه گراف، این متد را صدا بزنید. در غیر اینصورت، گراف قبلی و گراف جدید با هم نمایش داده می‌شوند و شکل بسیار شلوغی ایجاد می‌شود.
* در این پروژه هر خط از لاگ به منزله یک یال برای گراف هست و هر تمپلیت یک نود گراف را برای ما تشکیل می‌دهد.
  + 1. کدهای پیاده سازی کلاس گراف neo4j

1. ساختار و معماری فولدرهای پروژه به صورت زیر است:



روند اجرای برنامه، پردازش لاگ‌ها و تولید گراف به صورت زیر طی می‌شود:

from typing import Dict

class Neo4jHandler:

    graphDB\_Driver = None

    def \_\_init\_\_(self, config: Dict) -> None:

        uri = config["uri"]

        userName = config["userName"]

        password = config["password"]

        self.graphDB\_Driver = GraphDatabase.driver(

            uri, auth=(userName, password)

        )

    def create\_new\_node(self, node):

        create\_node = \

            "CREATE \n" + \

            f"({node}:node " + "{ name: " + f"\"{node}\"" + "})"

        with self.graphDB\_Driver.session() as graphDB\_Session:

            graphDB\_Session.run(create\_node)

        return create\_node

    def create\_new\_edge(self, node\_1, node\_2, edge, edge\_name):

        create\_edge = \

            "MATCH (u:node {name:" + f"'{node\_1}'" + "}), (r: node {name: " + \

            f"'{node\_2}'" + "}) " + \

            f"CREATE(u)-[:{edge}" + \

            " {source\_destination: " + f"'{edge\_name}'" + "}]->(r)"

        with self.graphDB\_Driver.session() as graphDB\_Session:

            graphDB\_Session.run(create\_edge)

        return create\_edge

    def clear\_graph(self):

        cqldelete1 = "match (a) -[r] -> () delete a, r"

        cqldelete2 = "match (a) delete a"

        with self.graphDB\_Driver.session() as graphDB\_Session:

            graphDB\_Session.run(cqldelete1)

            graphDB\_Session.run(cqldelete2)

در این قسمت در ادامه قسمت 2-3-2، گراف سیستم را به دست می‌آوریم. در ابتدا کلاسترهای خود را به صورت یک رابطه درمی آوریم. بدین منظور ابتدا نودهای موجود در کلاسترها را استخراج میکنیم و سپس با پیمایش کلاسترها، مشخص میکنیم که هر نود با چه نود دیگری در ارتباط است. در قطعه کد زیر nodes مشخص کننده نودها و connections مشخص کننده ارتباطات هر نود است. Nodes, connections مستقیما به neo4j داده می‌شوند.

@staticmethod

    def extractconnections(logs\_flows):

        nodes = []

        for i in range(len(logs\_flows)):

            for j in range(len(logs\_flows[i])):

                if logs\_flows[i][j] not in nodes:

                    nodes.append(logs\_flows[i][j])

        nodes = np.array(nodes)

        connections = []

        flow\_index = []

        for i in range(len(nodes)):

            temp\_connections = []

            temp\_flow\_index = []

            for k in range(len(logs\_flows)):

                for p in range(len(logs\_flows[k])):

                    if nodes[i] == logs\_flows[k][p] and p < len(logs\_flows[k]) - 1:

                        index\_of\_node = np.where(nodes == logs\_flows[k][p + 1])[0]

                        if index\_of\_node != i:

                            temp\_connections.append(index\_of\_node[0])

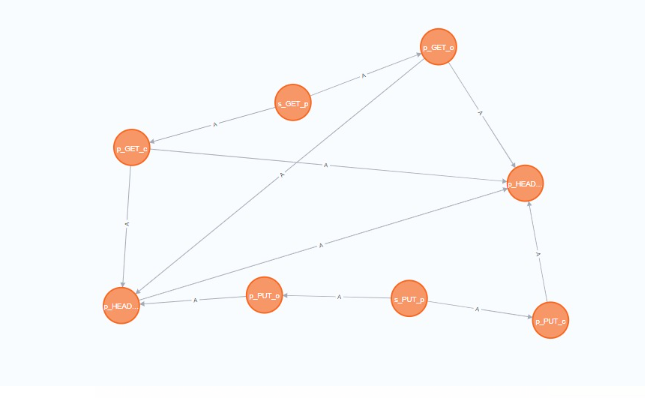
                            temp\_flow\_index.append(k)

            connections.append((np.array(temp\_connections)))

            flow\_index.append((np.array(temp\_flow\_index)))

        return nodes, connections, flow\_index

گراف سیستم در Neo4j به صورت زیر است:



شکل 39 : خروجی سیستم

* + 1. اضافه کردن programname، method و user-agent به عنوان ویژگی های نود در Neo4j

در این قسمت ویژگی‌های زیر را به یک نود در گراف خود اضافه میکنیم:

* Programname
* User-agent
* Method

به عنوان مثال در نمونه لاگ P-GET-a موارد فوق به صورت زیر مشخص می‌شوند :

* Programname:account-server
* User-agent: proxy-server
* Method: Get

برای این کار تابع create\_node را به صورت زیر اصلاح می‌کنیم :

def create\_new\_node(self, node):

        useragent = node.split("\_")[0]

        method = node.split("\_")[1]

        programname = node.split("\_")[2]

        if useragent == "p":

            useragent = "proxy\_server"

        elif useragent == "a":

            useragent = "account\_server"

        elif useragent == "o":

            useragent = "object\_server"

        elif useragent == "c":

            useragent = "container\_server"

        else:

            useragent = "s"

        if programname == "p":

            programname = "proxy\_server"

        elif programname == "a":

            programname = "account\_server"

        elif programname == "o":

            programname = "object\_server"

        elif programname == "c":

            programname = "container\_server"

        else:

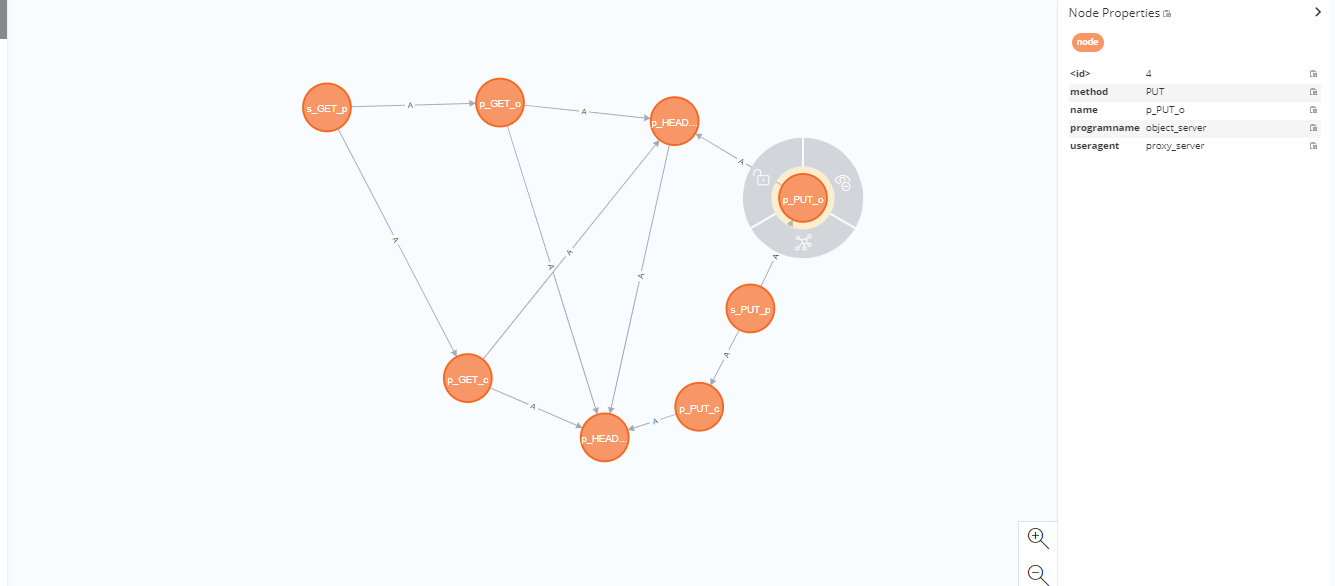
            programname = "s"

        create\_node = \

            "CREATE \n" + \

            f"({node}:node " + "{ name: " + f"\"{node}\" , programname: " + f"\"{programname}\" , method: " + f"\"{method}\" , useragent: " + f"\"{useragent}\"" + "})"

خروجی کد فوق در neo4j به صورت شکل زیر مشخص می‌شود:



شکل 40 : نمایش ویژگی ها در نود

1. فصل چهارم: سیستم کشف ناهنجاری
   1. معماری سوییفت و اهمیت کشف ناهنجاری

در ابتدا، یک بحث کوتاهی روی معماری سوییفت انجام میدهیم و یک مروری بر روی سرویس ها داریم. این قسمت مشخص میکند که مقصود ما از ناهنجاری جه مفهومی است. اجزای سوییفت همانطور که در فصل‌های پیشین بحث شد به صورت زیر است:

**سرور پروکسی:** بخشی که مسئول اتصال همه اجزای سوییفت به همدیگر است. با هر درخواست، پروکسی به دنبال محل حساب، کانتینر و یا شی جستجو کرده و درخواست را مطابق آن در مسیر درست خود هدایت می‌کند. در برخی مواقع پروکسی مسئول رمزگذاری و رمزگشایی اشیا نیز هست. همچنین تعداد زیادی از خطاها و عدم موفقیت‌ها در پروکسی مدیریت می‌شوند، مثلا اگر هنگام قرار دادن شی، سرور در دسترس نباشد، پروکسی از رینگ درخواست سرور جایگزین کرده و درخواست را به آن مسیر هدایت می‌کند. اشیایی که از یا به سرور شی منتشر می‌شوند، مستقیما از طریق پروکسی به یا از سمت کاربر هدایت می‌شوند.

**رینگ:** نماینده نگاشتی بین نام موجودیت‌های ذخیره شده بر روی دیسک و محل فیزیکی آن‌هاست. در مورد هر سیاست ذخیره سازی، رینگ‌های مختلفی برای حساب، کانتینر و اشیا وجود دارند. هر مولفه‌ای که قصد انجام عملیاتی روی شی، کانتینر یا حساب داشته باشد، باید با رینگ متناسب آن تعامل کند تا محل آن در کلاستر را پیدا کند.

**سرور شی:** سرور ذخیره سازی ساده که می‌تواند برای ذخیره سازی، بازیابی و حذف اشیای ذخیره شده روی دستگاه‌ها استفاده شود. اشیا با استفاده از مسیری که با نام هش شی و زمان عملیات تولید می‌شود، ذخیره می‌شوند.

**سرور کانتینر:** وظیفه اصلی سرور کانتینر این است که لیست بندی اشیا را مدیریت کند. این سرور اطلاعی از محل اشیا ندارد، بلکه فقط این که چه اشیایی داخل یک کانتینر خاص هستند. لیست‌های مذکور به صورت فایل دیتابیس sqlite ذخیره شده و مشابه خود اشیا در سراسر کلاستر تکرار می‌شوند.

**سرور حساب:** مشابه سرور کانتینر، با این تفاوت که وظیفه آن نگهداری و مدیریت لیست بندی کانتینر هاست.

با توجه به سرویس های معرفی شده، هر آن چه که گراف تولید شده در فصل پیش را پیروی نکند به عنوان ناهنجاری درنظر گرفته می‌شود. منبع لاگ‌های ما در این قسمت، لاگ های تولید شده توسط syslog است.

ابزار سوئیفت لاگ نسبتا مفصلی تولید می‌کند که می‌تواند برای موارد متعددی استفاده شود، از جمله: نظارت بر کارکرد کلاسترها، محاسبات مربوط به بهره‌وری، و ثبت سوابق حسابرسی[[40]](#footnote-40). لاگ‌های سوئیفت برای syslog ارسال می‌شوند و با توجه به سطح لاگ و امکانات syslog سازماندهی می‌شوند. تمام لاگ‌های مربوط به یک درخواست شناسه تراکنش یکسانی دارند. لاگ های موجود در syslog را ابتدا پردازش میکنیم و به فرمت زیر که یک دیکشنری است تبدیل می‌کنیم. در زیر چند نمونه از لاگ‌های مرتب شده آمده است:

[

  {

    "": 0,

    "log\_info": "-",

    "@version": 1,

    "facility": "local1",

    "bytes\_sent": 0,

    "bytes\_recvd": 3131,

    "request\_end\_time": "2022-05-31T09:45:51.316Z",

    "bytes\_recvd\_format": 3131,

    "object\_path": "folder8/file173.txt",

    "tags": "['swift', 'PROXY\_SER\_TRANSACTION']",

    "datetime": "31/May/2022/09/45/51",

    "procid": "-",

    "container\_path": "Mycontainer17",

    "protocol": 1,

    "@timestamp": "2022-05-31T09:45:51.317Z",

    "bytes\_per\_second": 14349.220898258478,

    "request\_method": "PUT",

    "port": 40082,

    "sysloghost": "m2-r1z1s1",

    "severity": "info",

    "account\_path": "AUTH\_test",

    "transaction\_id": "tx0f782efac508406091fbd-006295e3cf",

    "programname": "proxy-server",

    "user\_agent": "python-requests/2.18.4",

    "message": "172.29.0.1 172.29.0.1 31/May/2022/09/45/51 PUT /v1/AUTH\_test/Mycontainer17/folder8/file173.txt HTTP/1.0 201 - python-requests/2.18.4 AUTH\_tk56362d58f... 3131 - - tx0f782efac508406091fbd-00295e3cf - 0.2182 - - 1653990351.098684311 1653990351.316907883 0",

    "source": "-",

    "request\_start\_time": "2022-05-31T09:45:51.098Z",

    "ver": "v1",

    "remote\_addr": "172.29.0.1",

    "auth\_token": "AUTH\_tk56362d58f...",

    "client\_ip": "172.29.0.1",

    "referer": "-",

    "client\_etag": "-",

    "policy\_index": 0,

    "host": "r1z1s1",

    "type": "rsyslog",

    "status\_int": 201,

    "bytes\_sent\_format": 0,

    "headers": "-",

    "request\_time": 0.2182,

    "bytes\_sum": 3131,

    "device\_name": "",

    "request\_path": "",

    "partition\_number": "",

    "server\_pid": ""

  },

  {

    "": 1,

    "log\_info": "",

    "@version": 1,

    "facility": "local0",

    "bytes\_sent": "",

    "bytes\_recvd": "",

    "request\_end\_time": "",

    "bytes\_recvd\_format": "",

    "object\_path": "folder8/file173.txt",

    "tags": "['swift', 'OBJ\_SER\_TRANSACTION']",

    "datetime": "31/May/2022:09:45:51",

    "procid": "-",

    "container\_path": "Mycontainer17",

    "protocol": "",

    "@timestamp": "2022-05-31T09:45:51.282Z",

    "bytes\_per\_second": "",

    "request\_method": "PUT",

    "port": 41688,

    "sysloghost": "m6-r1z1s1",

    "severity": "info",

    "account\_path": "AUTH\_test",

    "transaction\_id": "tx0f782efac508406091fbd-006295e3cf",

    "programname": "object-server",

    "user\_agent": "proxy-server 264",

    "message": "172.29.0.2 - - [31/May/2022:09:45:51 +0000] \"PUT /sdc6/881/AUTH\_test/Mycontainer17/folder8/file173.txt\" 201 - \"PUT http://127.0.0.1:8082/v1/AUTH\_test/Mycontainer17/folder8/file173.txt\" \"tx0f782efac508406091fbd-006295e3cf\" \"proxy-server 264\" 0.0838 \"-\" 145 0",

    "source": "",

    "request\_start\_time": "",

    "ver": "",

    "remote\_addr": "172.29.0.2",

    "auth\_token": "",

    "client\_ip": "",

    "referer": "PUT http://127.0.0.1:8082/v1/AUTH\_test/Mycontainer17/folder8/file173.txt",

    "client\_etag": "",

    "policy\_index": 0,

    "host": "r1z1s1",

    "type": "rsyslog",

    "status\_int": 201,

    "bytes\_sent\_format": "",

    "headers": "",

    "request\_time": 0.0838,

    "bytes\_sum": "",

    "device\_name": "sdc6",

    "request\_path": "/sdc6/881/AUTH\_test/Mycontainer17/folder8/file173.txt",

    "partition\_number": 881,

    "server\_pid": 145

  },

  {

    "": 2,

    "log\_info": "",

    "@version": 1,

    "facility": "local0",

    "bytes\_sent": "",

    "bytes\_recvd": "",

    "request\_end\_time": "",

    "bytes\_recvd\_format": "",

    "object\_path": "",

    "tags": "['swift', 'CONT\_SER\_TRANSACTION']",

    "datetime": "31/May/2022:09:45:51",

    "procid": "-",

    "container\_path": "Mycontainer17",

    "protocol": "",

    "@timestamp": "2022-05-31T09:45:51.129Z",

    "bytes\_per\_second": "",

    "request\_method": "HEAD",

    "port": 40452,

    "sysloghost": "m8-r1z1s1",

    "severity": "info",

    "account\_path": "AUTH\_test",

    "transaction\_id": "tx0f782efac508406091fbd-006295e3cf",

    "programname": "container-server",

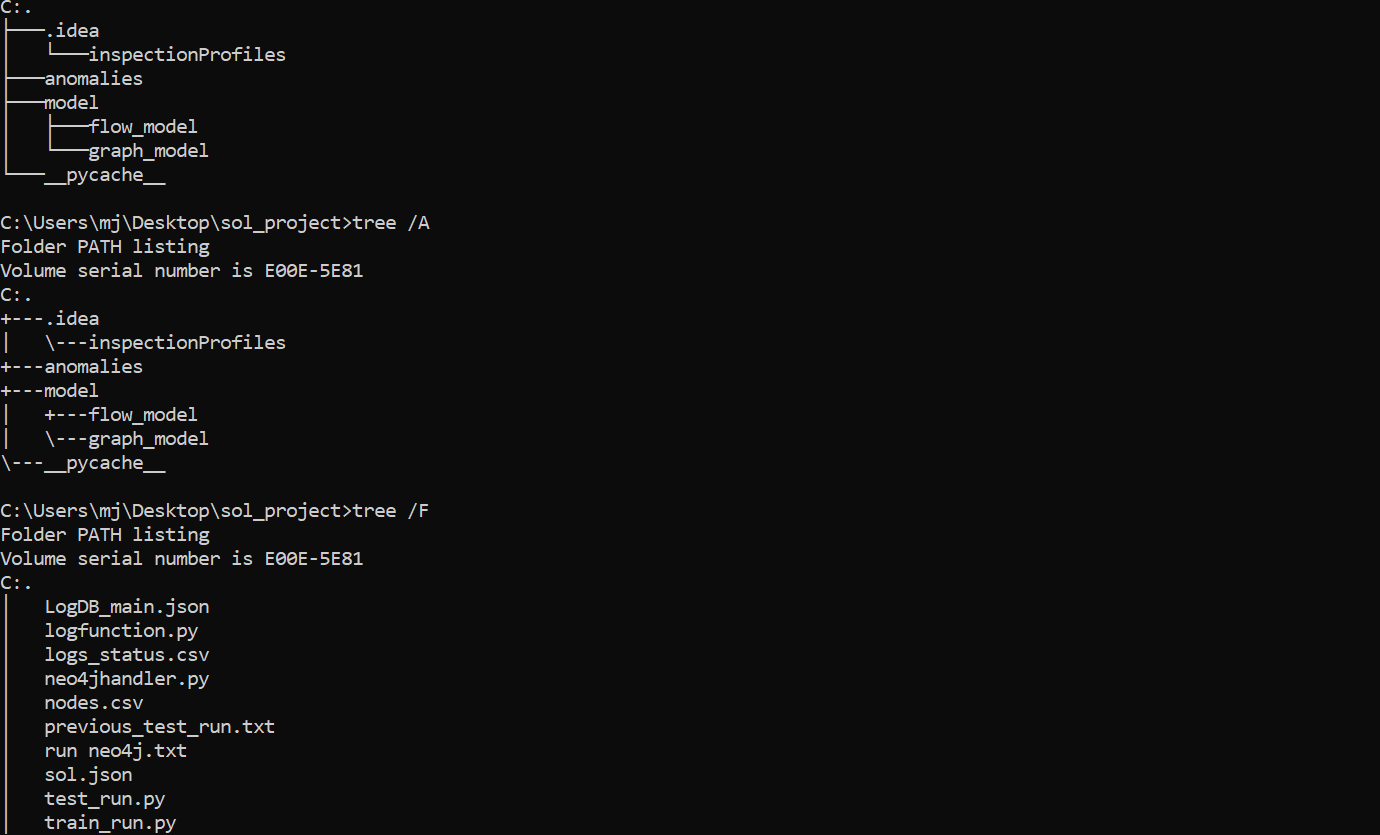
    "user\_agent": "proxy-server 264",

    "message": "172.29.0.2 - - [31/May/2022:09:45:51 +0000] \"HEAD

/sdc8/823/AUTH\_test/Mycontainer17\" 204 - \"HEAD

* + 1. الگوریتم کشف ناهنجاری

ساختار و معماری فولدرهای پروژه به صورت زیر است:



الگوریتم ناهنجاری پیشنهادی، موارد زیر را به عنوان لاگ ناهنجار درنظر می‌گیرد:

* توالی لاگ های یک تراکنش از نود استارت شروع نشود و یا در نود پایانی خاتمه نیابد.
* دو لاگ متوالی از یک تراکنش، فاصله زمانی بیشتر از 5 ثانیه داشته باشند.

لاگ آتی در توالی تراکنش متناظر با نود آتی در فلو گراف نباشد. قطعه کد زیر در ابتدا آرایه های connections، start و node را load می‌کند و سپس با پیروی از قوانین فوق لاگ های ناهنجار را تشخیص می‌دهد.

import pandas as pd

import os

import glob

from logfunction import LogsFunction

import numpy as np

import json

def run(fileName):

""" check and detect anomalies

:param filename: name of the json file

:return: an excel file with (logstatus.csv), in the excel file anomalies are determined with 1.

"""

*# load model*

connections = pd.read\_csv("./model/graph\_model/" + "connections.csv")

nodes = pd.read\_csv("./model/graph\_model/" + "nodes.csv")

start = pd.read\_csv("./model/graph\_model/" + "start.csv")

extension = fileName.split(".")[1]

**if** extension == 'json':

f = open(fileName)

json\_read = json.load(f)

print("start processing ...")

files = glob.glob('./anomalies/\*')

**for** f **in** files:

print(f)

os.remove(f)

new\_json = []

**for** i **in** range(len(json\_read)):

**if** json\_read[i]["user\_agent"] != "Swift":

new\_json.append(json\_read[i])

*# make standard logs (for example p-get-a)*

pure\_logs, transaction\_ids, datetime = LogsFunction().purifylogs(new\_json)

*# extract flows*

logs\_flows, datetime, tx = LogsFunction().extractFlows(pure\_logs, transaction\_ids, datetime)

*# sort logs*

sorted\_log\_flows = LogsFunction().sortlogs(logs\_flows, datetime)

*# detect anomalies*

count = np.zeros(len(sorted\_log\_flows), dtype=object)

temp\_tx = []

**for** i **in** range(len(sorted\_log\_flows)):

count[i] = 0

priority = []

**for** u **in** range(len(sorted\_log\_flows[i])):

**for** j **in** range(len(nodes)):

**if** sorted\_log\_flows[i][u] == nodes["0"][j]:

priority.append(j)

sorted\_priority = []

**for** b **in** range(len(priority)):

sorted\_priority.append(priority[len(priority) - 1 - b])

**if** start["0"][sorted\_priority[0]] == 0:

count[i] = 1

temp\_tx.append(tx[i][0])

**continue**

counter = np.zeros(len(sorted\_priority) - 1)

**for** k **in** range(len(sorted\_priority) - 1):

**for** m **in** range(len(connections[f"{priority[k]}"])):

**if** str(connections[f"{m}"][priority[k]]) == 'nan':

**continue**

**if** int(connections[f"{m}"][priority[k]]) == int(priority[k + 1]):

**if** counter[k] != 1:

counter[k] = counter[k] + 1

**if** (sum(counter) != len(sorted\_priority) - 1) or (sum(counter) == 0):

count[i] = 1

temp\_tx.append(tx[i][0])

**continue**

**else**:

count[i] = 0

temp\_tx.append(tx[i][0])

**continue**

temp\_tx = np.array(temp\_tx)

logs\_status = np.concatenate((count.reshape(-1, 1), temp\_tx.reshape(-1, 1)), axis=1)

df = pd.DataFrame(logs\_status)

*# save logs status in logs\_status.csv, 1 for anomalies*

filename = 'logs\_status.csv'

df.to\_csv(filename)

*# save anomalies in anomalies folder*

**for** i **in** range(len(tx)):

exact\_logs = []

**if** count[i]:

**for** j **in** range(len(json\_read)):

**if** json\_read[j]["transaction\_id"] == tx[i][0]:

exact\_logs.append(json\_read[j])

df = pd.DataFrame(exact\_logs)

filename = "./anomalies/" + str(i) + "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" + str(tx[i][0]) + '.csv'

df.to\_csv(filename)

print("processing finished")

print('Enter the filename:')

filename = input()

try:

extension = filename.split(".")[1]

**if** extension != "json":

print("not a json file")

exit(0)

run(filename)

except:

print(f"There is an error, please check the following items:\n")

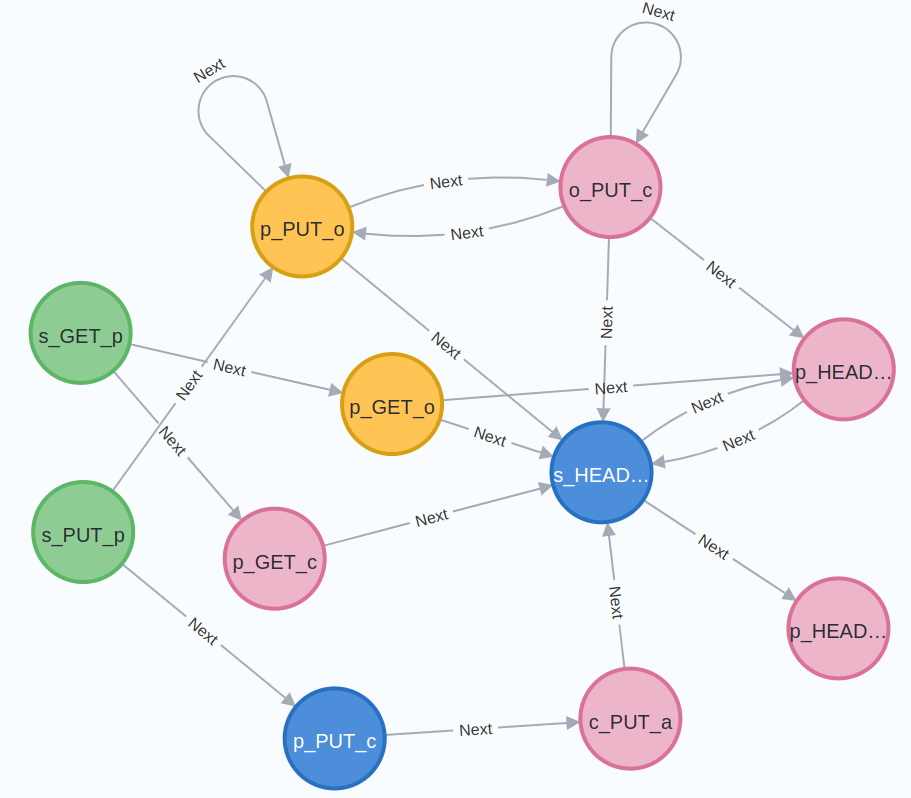
print("1 - The file must be exist in this directory\n")

print("2- The input file must be a json file. please check the extension of the file and make sure that the data "

"structure of file is json.")

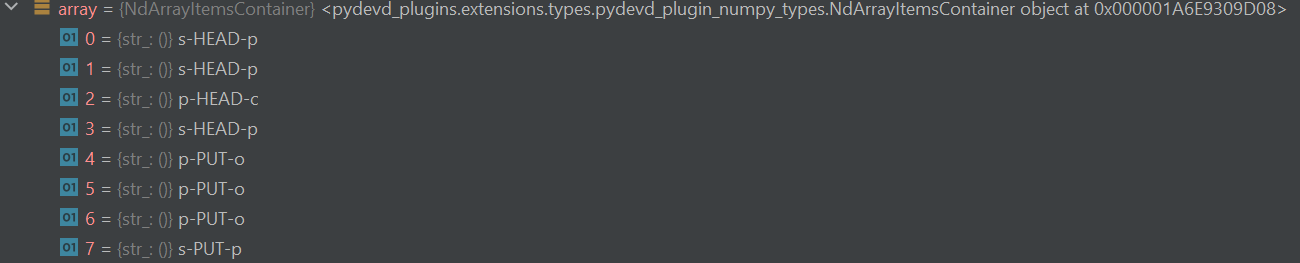
* + 1. بررسی چند نمونه لاگ ناهنجار

شکل نهایی گراف لاگ که ملاک پردازش و کشف ناهنجاری قرار میگیرد در شکل 42 آمده است:



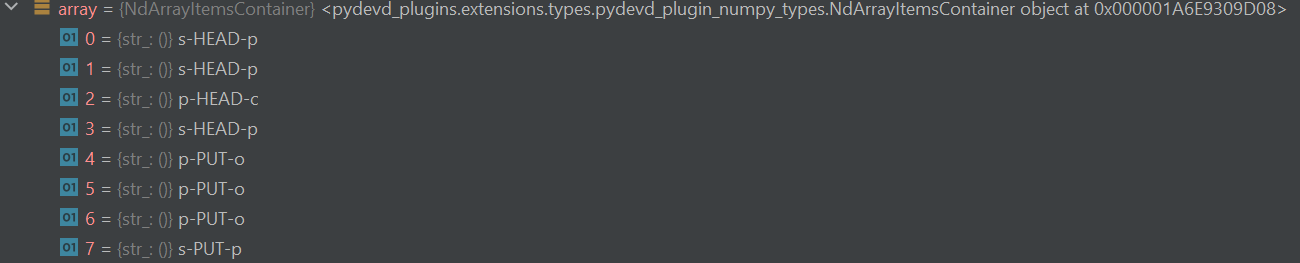
شکل 40 : گراف نهایی

نمونه لاگ مرتب شده و اصلاح شده طبق شکل 42 است.



شکل 41 : نمونه ناهنجاری

این مورد ناهنجاری اعلام شده است چون s-Head-p اتصال مجدد به خودش ندارد. نمونه دیگری هم از لاگ مرتب شده طبق شده 43 است.



شکل 42 : نمونه ناهنجاری

این مورد ناهنجاری اعلام شده است چون s-Head-p اتصال مجدد به خودش ندارد.

1. فصل پنجم: مرور کلی و کارهای آینده
   1. مقدمه

در این قسمت یک مرور کلی بر روی روش‌های مطالعه شده بر روی شناسایی گراف سلامت سیستم و کشف ناهنجاری در این سیستم ها می‌پردازیم. این قسمت از این نظر قابل اهمیت است که زمینه را برای کارهای آینده فراهم می‌کند.

* 1. روش های مطالعه شده در استخراج گراف سلامت سیستم

در یک سیستم ابری، همواره چالش سنجش عمکلرد صحیح سیستم وجود داشته است. بررسی این مساله در نوع خود بسیار سخت و دشوار است. ابزارهای مختلفی برای سنجش عملکرد سیستم در یک سیستم ابری وجود دارد که یکی از ابزارهای معروف grafana است. ابزار دیگری که این کار را انجام می‌دهد portainer است. در اینجا به طور خلاصه این دو ابزار را معرفی می‌کنیم.

* + 1. ابزار garafana

Grafana یک راهکار متن باز برای اجرای تحلیل و بررسی داده ها و رسم معیارهای منطقی روی انبوه داده ها است. همچنین از این راهکار به منظور مانیتور کردن اپلیکیشن ها نیز استفاده می شود که دارای داشبوردهای قابل شخصی سازی برای کمک به شما است. این ابزار به تمامی منابع داده ی ممکن متصل می شود و به طور معمول به دیتا بیس هایی مانندGraphite، Prometheus، Influx DB، ElasticSearch، MySQL، PostgreSQL و ... قابل ارجاع است. متن باز بودن گرافانا به ما این امکان را می دهد که برای ارتباط و همکاری با منابع داده مختلف بتوانیم پلاگین نویسی کنیم.

در سازمان ها طبیعتاً از Grafana برای اهداف نظارتی استفاده می کنند و در محیط پیش تولید برای بررسی خطاهای ظاهر شده و زمان آپ تایم سرورها. در محیط های توسعه که به عنوان مثال از داکر کانتینرها استفاده می کنند ممکن است نمونه ها به دلایل و مشکلاتی از کار بیافتند که باعث از کار افتادن کلی سیستم شود. در این سناریوها از طریق داشبورد گرافانا می توانیم این مشکلات را به راحتی ردیابی کنیم که کار ما را بسیار راحت تر خواهد کرد.

* + 1. ابزار portainer

این ابزار به خوبی یک GUI کامل در اختیار ما قرار می‌دهد. اما با این تفاوت که دیگر نیاز نیست حتما بر روی همان کامپیوتر سرویس دهنده داکر نصب شود. بلکه می‌تواند به سرویس دهنده‌های مختلف داکر متصل شود. این ابزار این امکان را دارد که به چند تا سرویس دهنده داکر متصل شود. با استفاده از این ابزار می‌توان به خوبی سرویس‌های بزرگ را پیاده سازی و از آنها استفاده کرد. در ضمن Portainer بر روی وب در دسترس می‌باشد از این رو می‌توان آن را بر روی سرورهایی که دارای UI و Desktop نیستند به خوبی کاربرد دارد.

* + 1. سایر روش‌های مانیتورینگ

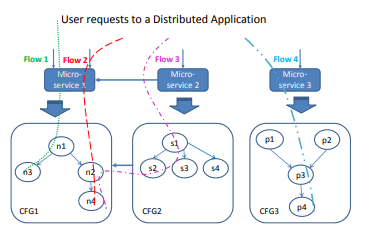
بعضی از مطالعات همواره بدنبال کشف رفتارسیستم و بررسی ناهنجاری از طریق یادگیری ماشین و با استفاده از دیتای جمع شده در سرور ابری هستند. یکی از ابزار مهم در این زمینه لاگ ها هستند. بسیاری از روش ها این کار رو در دو حالت انجام می‌دهند: 1- رسم گراف سلامت سیستم، 2- استخراج ناهنجاری براساس گراف سلامت سیستم. ناهنجاری در حقیقت هر انچه که مسیری در گراف پیدا نکند، تعریف می‌شود.[1] و [2] از این روش برای تشخیص ناهنجاری بهره ‌می‌برند.

استخراج گراف سلامت سیستم را می‌توان مشابه مساله network inference problem دانست. [3] و [4] برپایه این تعریف به حل این مساله پرداختند.

با توجه به پیچیده بودن این مساله، در بعضی از مطالعه ها فرض های برای ساده تر شدن این مساله انجام شده است. برای مثال در [5] و [6] و فرض شده که نودهای گراف از قبل مشخص هستند و مشارکت اصلی این مطالعه ها در تشخیص یال های بین این نودهاست.

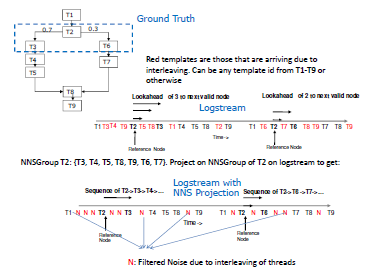
در بعضی مطالعه ها فرض شده است که قسمت متغیر یک لاگ در ابتدای لاگ است و طول این قسمت هم ناچیز است. مراجع [7] و [8] از این فرض استفاده کردند.

در دسته دیگری از مطالعه ها فرض شده است که لاگ های ما دارای transaction id هستند. این امر در شناسایی یک فرایند بسیار کمک کننده است. این مورد در مراجع [9] و [10] وجود دارد. به طور کلی یک سیستم توزیع شده را می‌توان به صورت زیر درنظر گرفت:



شکل 43 : سیستم توزیع شده

در مقاله [10] که یک مطالعه بزرگ در گروه تحقیقات شرکت IBM است، فرض شده که هیچ گونه دانش اولیه ای نسبت به لاگ ها وجود ندارد. این رویکرد تشخیص گراف سلامت سیستم را بسیار دشوار می‌کند. خلاصه این روش به شرح زیر است : هر لاگ در سیستم نرم‌افزاری به دو دسته تقسیم می‌شود: قسمت تغییرپذیر و قسمت ثابت. قسمت متغیر شامل موارد مانند آدرس پورت و زمان انجام لاگ است. جهت آن که به یک گراف کنترل جریان برسیم باید برای هر لاگ تعدادی پدر و فرزند تشکیل شود. برای تشکیل پدر و فرزند برای هر تمپلیت باید توالی زمانی لاگ‌ها درنظر گرفته شود. به عنوان مثال فرض کنید که بخواهیم برای یک تمپلیت پدر و فرزند استخراج کنیم. در شکل 44 این فرآیند نمایش داده شده است. دراین شکل، از یک روش آماری برای استخراج پدر و فرزندهای t2 استفاده شده است. برای آن‌که اساس این روش آماری توضیح داده شود ابتدا توالی لاگ‌های نمایش داده شده در شکل را درنظر بگیرید. تعدادی از این لاگ‌ها نویز هستند. با یک میانگین‌گیری ساده می‌توان متوجه شد که اکثر مواقع بعد از تمپلیت T1، تمپلیت T2 به وقوع می‌پیوندد. موارد نادر دیگری هم هستند که به عنوان نویز درنظر گرفته می‌شوند. با همین روش متوجه می‌شویم که بعد از T2 در 70 درصد موارد T3 و در 30 درصد موارد T6 به وقوع می‌پیوندد. موارد نادر دیگری هم هستند که به عنوان نویز درنظر گرفته می‌شوند. به همین سادگی پدر و فرزند تمپلیت T2 استخراج می‌شود. اما سوالی که یه وجود می‌آید این است: با چه حد آستانه‌ای موارد نویزی را تشخیص دهیم؟ برای پاسخ به این سوال از یک قاعده سرانگشتی استفاده می‌کنیم و آن هم این نکته است: اگر درصد حضور زیر 10 درصد باشد، آن مورد به عنوان نویز درنظر گرفته شود.



شکل 44 : تشکیل پدر و فرزند برای هر تمپلیت

پس از این مرحله گراف های مربوط به هر مرحله با هم مرج می‌شوند.اگر شباهت بین دو تمپلیت از نظر متنی بالای یک حد آستانه باشد دو گراف با همدیگر مرج می‌شوند و بین پدرها و فرزنداشنون اجتماع گرفته می‌شود. اما اگر این شباهت فقط بین فرزندان و یا پدرها باشد، دو گراف از محل آن شباهت بایکدیگر مرج می‌شوند این مرج شدن به همین صورت بین هر دو گراف تکرار می‌شود تا در نهایت به یک گراف کلی برسیم. گراف به دست آمده همان گراف کنترل جریان برای لاگ‌ها است. معیار شباهت هم جاکارد است. در یک سیستم نرم‌افزاری پس از تشکیل گراف کنترل جریان و گراف هر تمپلیت، دو حالت می‌تواند نشانه ناهنجاری باشد:

الف) در یک بازه زمانی مشخص، فرزندان یک تمپلیت مشاهده نشوند. اگر همه فرزندان مشاهده نشوند احتمال ناهنجاری بیشتر از حالتی است که یک یا تعدادی از فرزندان مشاهده نشوند.

ب) توزیع فرزندان یا پدران هر تمپلیت در یک بازه زمانی تغییر کند.

در [11]، گراف حالت با مدل متفاوتی ساخته می‌شود و روی لاگهای هدوپ این پروسه تست می‌شود. برای ساخت گراف حالت از لاگ‌های خام استفاده می‌شود. در ابتدای کار نیاز است که هر لاگ خام آنالیز شود و دو قسمت متغیر و اصلی از آن استخراج شود. پس از این مرحله، ساخت گراف حالت آغاز می‌شود. هر گراف حالت سه قسمت اصلی دارد: موجودیت، حالت، رخداد. یال‌ها به دو دسته فضایی و زمانی تقسیم می‌شوند. هر یال فضایی برای ایجاد یک رابطه بین یک موجودیت با یک رخداد و یا یک موجودیت با یک حالت استفاده می‌شود. هیچ‌گاه، یک رخداد با یک رخداد رابطه مستقیم ندارد و باید یک حالت و یا یک رخداد بین دو رخداد واسطه شود.

الف) حالت: این موارد در سیستم‌های نرم‌افزاری موارد مشخصی هستند و شامل مواردی مانند warning، error، run می‌شوند.

خلاصه ای از رویکردهای مختلف در مساله پردازش گراف در جدول زیر جمع اوری شده است:

|  |  |
| --- | --- |
| رویکرد | مرجع |
| روش هیبرید در تشخیص گراف سلامت سیستم | [1] و [2] |
| حل مساله با دید network inference problem | [3] و [4] |
| فرض های ساده کننده در حل مساله | [5] و [6] و [7]  [8] ، [9] |
| حل مساله بدون فرض ساده کننده | [10] |

|  |  |  |  |
| --- | --- | --- | --- |
| شماره مقاله | تکنیک | دیتاست | متودارزیایی |
| مقاله شماره 1 | رسم گراف + کشف ناهنجاری با درنظر گرفتن فرض وجود ترنزکشن آیدی بر روی گراف های با فرمت مشخص و ساده سازی شده | لاگ های هدوپ | False positive |
| مقاله شماره 2 | رسم گراف + کشف ناهنجاری با درنظر گرفتن فرض کوتاه بودن طول قسمت متغیر لاگ | لاگ های هدوپ | False positive |
| مقاله شماره 3 | کشف ناهنجاری در لاگ ها با استفاده از تکنیک های بیزین در network inference problem | SILK + هدوپ | False positive |
| مقاله شماره4 | کشف ناهنجاری در لاگ ها با استفاده از فرض بولین در network inference problem | SILK + اسپارک | False positive |
| مقاله شماره 5 | کشف ناهنجاری در لاگ ها با استفاده از فرض مشخص بودن نود ها از قبل و فرض وجود ترنزشکن آیدی و استخراج کانکشن ها با متود ساخت گراف کامل از زیرگراف ها | لاگ های ویندوز + لاگ های APACHE ERROR | تعداد ناهنجاری درست به نسبت کل |
| مقاله شماره 6 | بهبود مثال شماره پنج در کلاسترینگ گروه لاگ های مشابه با تغییر متود شباهت از فازی به جاکارد | لاگ های هدوپ | تعداد ناهنجاری درست به نسبت کل |
| مقاله شماره 7 | استخراج ویژگی + PCA جهت کشف ناهنجاری | لاگ های هدوپ | False positive |
| مقاله شماره 8 | کلاستر کردن لاگ ها براساس معیار edit distance based  به این صورت که شباهت هر دو لاگ اگر بالای ترشلد مشخصی باشد در یک گروه قرار می گیرند | لاگ های هدوپ | تعداد ناهنجاری درست به نسبت کل |
| مقاله شماره 9 | استفاده از متود partiation by search bijection  در استخراج کلاسترها | لاگ های هدوپ | تعداد ناهنجاری درست به نسبت کل |
| مقاله شماره 10 | استخراج گراف سلامت + ناهنجاری بدون هیچ گونه فرض اولیه با استفاده از متود پدر و فرزند در استخراج توالی و کلاسترینگ با معیار جاکارد و کشف ناهنجاری با پیشمایش آرایه ای کانکشن های گراف | لاگ های هدوپ | تعداد ناهنجاری درست به نسبت کل |

در هر کدام از این مقاله ها، اعدادی به عنوان دقت این الگوریتم ها براساس معیار استفاده شده در خود مقاله ذکر شده است. اما نکته قابل تامل این است که این مقاله ها نتایج خود را با هم مقایسه نکرده اند.

1. فصل ششم: مراجع

[1] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. In SIGOPS Operation Systems Review, 2010.

[2] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. In SIGOPS Operation Systems Review, 2010.

[3] B. Abrahao, F. ChieriChetti, R. Kleinberg, and A. Panconesi. Trace complexity of network inference. In KDD, 2013.

[4] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. In KDD, 2010.

[5] W. V. der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. In TKDE, 2004.

[6] C. W. Gunther and W. M. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In BPM, 2007

[7] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large scale system problems by mining console logs. In ICML, 2010.

[8] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In ICDM, 2009.

[9] A. Makanju, A. Z. Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In KDD, 2009.

[10] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining program workflow from interleaved traces. In KDD, 2010.

1. sonar [↑](#footnote-ref-1)
2. Reliability Remediation Effort [↑](#footnote-ref-2)
3. swift [↑](#footnote-ref-3)
4. Security Remediation Effort [↑](#footnote-ref-4)
5. technical debt [↑](#footnote-ref-5)
6. density [↑](#footnote-ref-6)
7. lines [↑](#footnote-ref-7)
8. comment lines [↑](#footnote-ref-8)
9. NCLOC : Non Comment Lines Of Code [↑](#footnote-ref-9)
10. Cyclomatic Complexity [↑](#footnote-ref-10)
11. Cognitive Complexity [↑](#footnote-ref-11)
12. break [↑](#footnote-ref-12)
13. effort [↑](#footnote-ref-13)
14. factor analysis [↑](#footnote-ref-14)
15. missing data [↑](#footnote-ref-15)
16. outlier [↑](#footnote-ref-16)
17. outlier [↑](#footnote-ref-17)
18. Artificial Neural Networks - ANN [↑](#footnote-ref-18)
19. Control flow graph (CFG) [↑](#footnote-ref-19)
20. Log [↑](#footnote-ref-20)
21. Variant [↑](#footnote-ref-21)
22. Invariant [↑](#footnote-ref-22)
23. Port address [↑](#footnote-ref-23)
24. Predecessor [↑](#footnote-ref-24)
25. successor [↑](#footnote-ref-25)
26. Template [↑](#footnote-ref-26)
27. Threshold [↑](#footnote-ref-27)
28. Text similarity [↑](#footnote-ref-28)
29. Jaccard similarity [↑](#footnote-ref-29)
30. Anomaly detection [↑](#footnote-ref-30)
31. State graph [↑](#footnote-ref-31)
32. entity [↑](#footnote-ref-32)
33. state [↑](#footnote-ref-33)
34. event [↑](#footnote-ref-34)
35. Spatial [↑](#footnote-ref-35)
36. temporal [↑](#footnote-ref-36)
37. Spatial edge [↑](#footnote-ref-37)
38. Temporal edge [↑](#footnote-ref-38)
39. <https://hub.docker.com/_/neo4j/> [↑](#footnote-ref-39)
40. [↑](#footnote-ref-40)