دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

عنوان گزارش فارسی

زیرعنوان گزارش

نویسندگان:

محمدسینا اله‌کرم	**محمد اکبرپورجنت**	**محمد شمس الدینی**	**نام نویسنده چهارم**
نام نویسنده پنجم	**نام نویسنده ششم**	**نام نویسنده هفتم**	**نام نویسنده هشتم**
نام نویسنده نهم	**نام نویسنده دهم**	**نام نویسنده یازدهم**	**نام نویسنده دوازدهم**
نام نویسنده سیزدهم	**نام نویسنده چهاردهم**	**نام نویسنده پانزدهم**	

استاد راهنما: دکتر محمدهادی علائیان

۹ آبان ۱۴۰۴

چکیده

این قسمت شامل چکیده گزارش است. چکیده باید خلاصه‌ای جامع از محتوای گزارش را ارائه دهد و شامل موارد زیر باشد:

- هدف از انجام پروژه یا تحقیق
- روش‌های استفاده شده
- نتایج اصلی به دست آمده
- نتیجه‌گیری کلی

کلیدواژه‌ها: لاتک، فارسی، ،xepersian گزارش، قالب

فهرست مطالب

فهرست تصاویر

فصل ۱

فرایندهای مهندسی نرم‌افزار و چرخه‌های تکامل تا پیدایش

این فصل شامل مقدمه‌ای بر موضوع گزارش است.

در این فصل به مقدمه ای بر موضوع گزارش می پردازیم.

- هدف از انجام پروژه
- روش‌های استفاده شده
- نتایج اصلی به دست آمده
- نتیجه‌گیری کلی

۱.۱ پیش‌زمینه

در این بخش پیش‌زمینه و انگیزه انجام پروژه توضیح داده می‌شود.

۱.۱.۱ تاریخچه

تاریخچه موضوع مورد بررسی در این قسمت آورده می‌شود. می‌توان به کارهای گذشته و پیشرفت‌های صورت گرفته اشاره کرد.

۲.۱.۱ اهمیت موضوع

توضیح اهمیت و کاربردهای موضوع در این قسمت قرار می‌گیرد. برای مثال:

- کاربرد در صنعت
- کاربرد در تحقیقات علمی
- کاربرد در زندگی روزمره

می‌توان از [۱۸] برای ارجاع به منابع استفاده کرد.

۲.۱ اهداف پروژه

اهداف اصلی این پروژه عبارتند از:

۱. بررسی و مطالعه موضوع اصلی
۲. طراحی و پیاده‌سازی راه‌حل پیشنهادی
۳. ارزیابی و مقایسه نتایج
۴. ارائه پیشنهادات برای کارهای آینده

۱.۲.۱ اهداف کوتاه‌مدت

اهداف کوتاه‌مدت شامل موارد زیر است:

- مطالعه ادبیات موضوع
- آشنایی با ابزارها و تکنیک‌های مورد نیاز

۲.۲.۱ اهداف بلندمدت

اهداف بلندمدت شامل:

- توسعه یک سیستم کامل
- انتشار نتایج در مجلات معتبر

۳.۱ ساختار گزارش

این گزارش در چندین فصل سازماندهی شده است:

فصل ۱ شامل مقدمه و اهداف پروژه است

فصل ۲ به بررسی مبانی نظری و کارهای مرتبط می‌پردازد

فصل ۳ نتایج و پیشنهادات را ارائه می‌دهد

در پایان نیز منابع و مراجع استفاده شده آورده شده است.

فصل ۲

مشکلات مطرح در چرخه‌های توسعه و تکامل نرم‌افزار

در این فصل به بررسی مبانی نظری و کارهای انجام شده قبلی می‌پردازیم.

۱.۲ مفاهیم پایه

در این بخش مفاهیم و تعاریف پایه‌ای ارائه می‌شود.

۱.۱.۲ تعاریف اولیه

تعریف ۱.۲. یک **تعریف نمونه** عبارت است از...

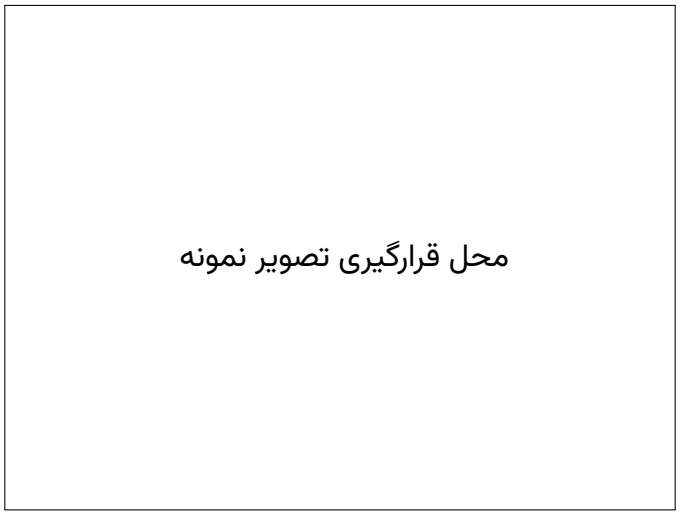
مثال ۱.۲. مثالی برای توضیح بهتر مفهوم: می‌توان فرمول ریاضی نوشت: $f(x) = x^۲ + ۲x + ۱$

۲.۱.۲ قضایای اساسی

قضیه ۱.۲. اگر a و b دو عدد حقیقی باشند، آنگاه:

$$(a + b)^۲ = a^۲ + ۲ab + b^۲$$

می‌توان به شکل‌ها نیز اشاره کرد، مانند شکل ۱.۲.



شکل ۱.۲: یک شکل نمونه برای نمایش

۲.۲ مرور ادبیات

مرور کارهای انجام شده در این حوزه:

۱.۲.۲ کارهای پیشین

محققان مختلفی در این زمینه فعالیت کرده‌اند. برای مثال:

- [۱۸]: ارائه روش نوین برای حل مسئله
- کارهای دیگر در این زمینه

۲.۲.۲ مقایسه روش‌ها

جدول زیر مقایسه‌ای بین روش‌های مختلف ارائه می‌دهد:

همانطور که در جدول ۱.۲ مشاهده می‌شود، هر روش مزایا و معایب خود را دارد.

۳.۲ روش‌های موجود

در این بخش روش‌های موجود برای حل مسئله بررسی می‌شود.

جدول ۱.۲: مقایسه روش‌های مختلف

روش	دقت	سرعت	پیچیدگی
روش الف	بالا	متوسط	کم
روش ب	متوسط	بالا	متوسط
روش ج	بالا	کم	زیاد

۱.۳.۲ روش اول

توضیحات مربوط به روش اول:

۱. گام اول: تعریف مسئله

۲. گام دوم: جمع‌آوری داده‌ها

۳. گام سوم: پردازش و تحلیل

۴. گام چهارم: ارائه نتایج

۲.۳.۲ روش دوم

روش دوم رویکرد متفاوتی دارد:

الگوریتم پیشنهادی به شکل زیر است:

```
def algorithm(data):
    result = []
    for item in data:
        if item > threshold:
            result.append(item)
    return result
```

Listing 2.1:

۳.۳.۲ مزایا و معایب

هر یک از روش‌های ذکر شده مزایا و معایب خاص خود را دارند که باید در انتخاب روش مناسب مدنظر قرار گیرند.

فصل ۳

DevOps و نقش آن در فرایند تکامل نرم‌افزار

۱.۳ نتایج

۲.۳ مقدمه و تعریف DevOps

۱.۳ مقدمه و تعریف DevOps

DevOps یک فرهنگ، فلسفه و مجموعه‌ای از روش‌ها و ابزارها است که هدف اصلی آن، یکپارچه‌سازی و خودکارسازی فرآیندهای بین تیم‌های توسعه نرم‌افزار (Development) و عملیات فناوری اطلاعات (Operations) است. در مدل سنتی، این دو تیم جدا از هم عمل می‌کردند که منجر به کندی، خطاهای بیشتر و هماهنگی دشوار می‌شد. ظهور DevOps پاسخی به این چالش‌ها بود تا با ایجاد همکاری و مسئولیت مشترک، شکاف بین ساخت نرم‌افزار و اجرای پایدار آن را از بین ببرد. در نهایت، DevOps به سازمان‌ها این توانایی را می‌دهد که نرم‌افزارها را سریع‌تر، قابل‌اطمینان‌تر و با کیفیت بالاتر در اختیار کاربران قرار دهند.

۳.۳ بحث و تفسیر

۴.۳ فلسفه DevOps و ارتباط آن با Agile

۲.۳ فلسفه DevOps و ارتباط آن با Agile

فلسفه DevOps بر پایه اصولی استوار است که فرهنگ همکاری، خودکارسازی و بهبود مستمر را ترویج می‌دهد. این فلسفه را می‌توان در "حلقه بی‌پایان" عملیات DevOps (که شامل مراحل برنامه‌ریزی، توسعه، استقرار و نظارت است) و همچنین در "سه راهی" معروف آن (جریان ،(Flow) بازخورد -Feed) (back و یادگیری مستمر Continuous) ((Learning خلاصه کرد.

ارتباط DevOps با متدولوژی Agile بسیار عمیق است. Agile بر انعطاف‌پذیری، تحویل تدریجی و پاسخگویی به تغییرات در طول فرآیند توسعه تأکید دارد. DevOps این فلسفه را گسترش می‌دهد و آن را به فرآیند استقرار و عملیات پس از توسعه تسری می‌بخشد. در حقیقت، DevOps مکمل Agile است؛ در حالی که Agile سرعت و کیفیت توسعه را افزایش می‌دهد، DevOps تضمین می‌کند که این تغییرات سریع می‌توانند به صورت ایمن و پایدار در محیط تولید مستقر شوند. بنابراین، می‌توان DevOps را به عنوان ادامه طبیعی و ضروری جنبش Agile در نظر گرفت که تمرکز آن بر روی کل چرخه عمر نرم‌افزار است.

۵.۳ پیشنهادات برای کارهای آینده

۶.۳ چرخه عمر DevOps

۳.۳ چرخه عمر DevOps

چرخه عمر DevOps یک فرآیند تکراری و مستمر است که مراحل مختلفی از ایده تا تحویل نرم‌افزار و نظارت بر آن را در بر می‌گیرد. این چرخه با استفاده از ابزارهای خودکار به هم پیوسته، جریان ارزش را سریع و کارآمد می‌کند.

- **برنامه‌ریزی (Plan)**

در این فاز اولیه، اهداف پروژه تعریف، وظایف زمان‌بندی و پیشرفت کار رهگیری می‌شود. این

مرحله تضمین می‌کند که همه اعضای تیم از اهداف کسب‌وکار و برنامه‌های فنی آگاه هستند.
ابزارها: از ابزارهایی مانند Jira برای ردیابی Issues و مدیریت پروژه و Confluence برای مستندسازی و همکاری استفاده می‌شود.

- **توسعه (Code)**

توسعه‌دهندگان در این مرحله نرم‌افزار را می‌نویسند. برای اطمینان از سازگاری و قابلیت تکرار محیط‌های توسعه، از ابزارهای خاصی استفاده می‌شود.
ابزارها: Docker برای بسته‌بندی نرم‌افزار در کانتینرهای سبک و قابل حمل، Kubernetes برای مدیریت و خودکارسازی این کانتینرها، و Puppet & Ansible برای مدیریت پیکربندی و خودکارسازی زیرساخت به کار می‌روند.

- **یکپارچه‌سازی مستمر (Continuous Integration)**

این تمرین شامل ادغام مکرر کد نوشته‌شده توسط تمام توسعه‌دهندگان به یک ریپازیتوری مشترک است. پس از هر ادغام، فرآیندهای ساخت و تست به طور خودکار اجرا می‌شوند تا خطاها در اسرع وقت شناسایی شوند. CI تضمین می‌کند که کدها به طور مداوم با یکدیگر یکپارچه شده و از بروز تعارضات بزرگ در آینده جلوگیری می‌کند.

- **تحویل مستمر (Continuous Delivery)**

CD گام بعدی پس از CI است. این تمرین تضمین می‌کند که پس از هر ادغام موفقیت‌آمیز کد، می‌توان نرم‌افزار را در هر لحظه و با کمترین تلاش به صورت دستی در محیط تولید منتشر کرد. در تحویل مستمر، فرآیند استقرار تا مرحله نهایی خودکار است، اما انتشار نهایی در محیط تولید به صورت دستی و با تأیید یک انسان انجام می‌شود.

- **استقرار مستمر (Continuous Deployment)**

این پیشرفته‌ترین مرحله است که در آن، هر تغییری که از تست‌ها در مراحل CI/CD موفقیت‌آمیز عبور کند، به طور خودکار در محیط تولید مستقر می‌شود. در این مدل، هیچ مداخله دستی در فرآیند استقرار وجود ندارد و انتشار نرم‌افزار به یک رویداد عادی و روزمره تبدیل می‌شود. این امر سرعت ارائه ارزش به کاربر نهایی را به حداکثر می‌رساند.
ابزارهای CI/CD: از ابزارهایی مانند Jenkins، GitHub Actions/GitLab CI/CD و CircleCI برای خودکارسازی کامل خط لوله از یکپارچه‌سازی تا استقرار استفاده می‌شود.

- **نظارت و بازخورد (Monitoring & Feedback)**

پس از استقرار نرم‌افزار در محیط تولید، عملکرد آن تحت نظارت دقیق قرار می‌گیرد تا از پایداری و سلامت سرویس اطمینان حاصل شود. داده‌های مربوط به عملکرد برنامه، زیرساخت و تجربه

کاربر جمع‌آوری و تجزیه و تحلیل می‌شوند. این داده‌ها به صورت یک حلقه بازخورد (Feed-back Loop) به تیم‌های توسعه و برنامه‌ریزی بازمی‌گردند تا برای بهبود مستمر محصول و رفع مشکلات در چرخه‌های بعدی مورد استفاده قرار گیرند.

ابزارها: Grafana برای تجسم متریک‌ها، Stack Elastic برای مدیریت و تحلیل لاگ‌ها، Prometheus برای مانیتورینگ و هشدار و Sentry برای ردیابی خطاها در لحظه استفاده می‌شوند.

فصل ۴

چرایی نیاز به بازطراحی در پیاده‌سازی نرم‌افزار

۱.۴ مقدمه

سیستم‌های نرم‌افزاری، برخلاف دارایی‌های فیزیکی که دچار فرسایش مکانیکی می‌شوند، به مرور زمان کارایی خود را در انطباق با واقعیت‌های تجاری و بستر فناورانه از دست می‌دهند. این پدیده، که اغلب به آن «کهنگی نرم‌افزاری» گفته می‌شود، منجر به افزایش فزاینده در هزینه‌های عملیاتی و نگهداری می‌گردد. برآوردها نشان می‌دهد که نگهداری نرم‌افزار به‌عنوان پرهزینه‌ترین فاز چرخه حیات نرم‌افزار، تقریباً ۶۰ درصد از کل تلاش‌های صورت گرفته در این چرخه را به خود اختصاص می‌دهد.

سازمان‌ها در محیط‌های رقابتی و نظارتی امروز، تحت فشار مستمر برای افزایش چابکی و پاسخگویی به تغییرات بازار، مقررات جدید، و نیازهای در حال تحول کاربران قرار دارند. زمانی که سیستم‌های قدیمی (Legacy Systems) به مانعی برای نوآوری تبدیل می‌شوند و بخش نامتناسبی از بودجه را مصرف می‌کنند، بازمهندسی (Reengineering) به یک ضرورت استراتژیک تبدیل می‌گردد. هدف از بازمهندسی، نه صرفاً تولید ویژگی‌های جدید، بلکه بازیابی و طولانی کردن عمر سیستم‌های حیاتی است، ضمن کاهش هزینه‌های بالای نگهداری.

بازمهندسی، اساساً یک سرمایه‌گذاری در مدیریت ریسک و بهینه‌سازی مالی محسوب می‌شود. زمانی که هزینه‌های نگهداری بیش از نیمی از بودجه توسعه را می‌بلعد، این هزینه عملاً منابعی را که می‌توانست صرف نوآوری شود، از بین می‌برد (هزینه فرصت). بنابراین، بازمهندسی به عنوان راهکاری برای تثبیت سازمانی، کاهش ریسک‌های شکست سیستمی، و تضمین انطباق با قوانین، بر تحویل ویژگی‌های فوری اولویت می‌یابد.

۲.۴ تعریف بازطراحی (Redesign) / (Reengineering

بازمهندسی نرم‌افزار، فرآیند بررسی و تغییر یک سیستم موجود با هدف پیاده‌سازی آن در یک فرم جدید یا تطبیق داده شده است. این فرآیند از نرم‌افزار و مستندات موجود استفاده می‌کند تا نیازمندی‌ها و طراحی سیستم هدف را تولید کند.

۱.۲.۴ بازمهندسی در برابر مهندسی رو به جلو

برخلاف مهندسی رو به جلو (Forward Engineering) که با یک سند مشخصات تعریف‌شده آغاز می‌شود، بازمهندسی با سیستم موجود به‌عنوان «مشخصات» خود آغاز شده و از طریق فرآیندهای درک و تبدیل، سیستم هدف را استخراج می‌کند.

۲.۲.۴ مهندسی معکوس (بازیابی طراحی)

مرحله حیاتی که بازمهندسی را تعریف می‌کند، بازیابی طراحی یا مهندسی معکوس (Reverse Engineering) است. این مرحله برای بازیابی منطق و چرایی تصمیمات معماری از دست‌رفته که در طول پیاده‌سازی اولیه اتخاذ شده‌اند، ضروری است. قبل از شروع هرگونه کار فنی، زمینه و هدف بازمهندسی باید در چارچوب اهداف کلان سازمانی تعریف شود.

۳.۴ دلایل اصلی نیاز به بازطراحی

۱.۳.۴ تغییر نیازمندی‌ها

تغییر در نیازمندی‌ها (ناشی از تکامل نیازهای کاربر یا تغییر بازار و مقررات) اجتناب‌ناپذیر است و ایجاد تغییرات دیرهنگام می‌تواند هزینه‌های توسعه را تا ۳۰ درصد افزایش دهد. استراتژی دفاعی، طراحی معماری بر اساس تجزیه مبتنی بر نوسان (Volatility-Based Decomposition) است. این اصل مستلزم کپسوله‌سازی اجزای مستعد تغییر برای جلوگیری از نشت تغییرات در سراسر سیستم و افزایش مقاومت در برابر انحراف ویژگی است. در غیر این صورت، سیستم با بدهی معماری روبه‌رو می‌شود.

۲.۳.۴ فناوری‌های جدید

پذیرش فناوری‌های جدید نیروی محرکه قوی برای بازمهندسی است و اغلب برای رفع محدودیت‌های عملکردی و مقیاس‌پذیری سیستم‌های قدیمی ضروری است. این شامل گذار به برنامه‌های ابرمحور (Cloud-Native) و معماری میکروسرویس‌ها می‌شود. زیرساخت‌های قدیمی IT اغلب نیازمند سفارشی‌سازی‌های گسترده و راه‌حل‌های میان‌افزار پیچیده هستند تا قابلیت همکاری با ابزارهای دیجیتال جدید تضمین شود.

۳.۳.۴ ضعف معماری اولیه

نیاز به بازطراحی اغلب ریشه در پذیرش الگوهای ضدطراحی (Anti-Patterns) دارد. بدنام‌ترین آن، الگوی «توپ گلی بزرگ» (Big Ball of Mud - BBoM) است که در آن ساختار سیستم فاقد تفکیک مسئولیت‌ها و سازماندهی مشخص است. این وضعیت باعث انباشت بدهی فنی شده و اصلاح سیستم را دشوار می‌کند.

۴.۳.۴ انباشت بدهی فنی

بدهی فنی، هزینه استعاری تصمیم‌های کوتاه‌مدت است. همانند بدهی مالی، بهره‌مند است و با گذشت زمان رشد می‌کند. بر اساس گزارش‌ها، حدود ۴۲٪ از زمان توسعه‌دهندگان صرف مقابله با بدهی فنی می‌شود. زمانی که بدهی فنی از ۵۰٪ ارزش فناوری یک سیستم فراتر رود، بازمهندسی کامل از نظر اقتصادی توجیه‌پذیر است.

۴.۴ مراحل بازطراحی نرم‌افزار

فرآیند بازمهندسی نرم‌افزار یک روش‌شناسی ساختاریافته است که از تحلیل سیستم موجود آغاز شده و تا پیاده‌سازی استراتژی‌های مهاجرت با کمترین ریسک ادامه می‌یابد.

۱.۴.۴ تحلیل سیستم فعلی

ارزیابی جامع سیستم فعلی گام اول است تا مشخص شود کدام بخش‌ها ارزش حفظ کردن دارند. این تحلیل شامل سنجش شاخص‌هایی مانند زمان پاسخ، درصد در دسترس بودن (uptime) و آزمون بار اوج (Peak Load Testing) است. همچنین ارزیابی هزینه کل مالکیت (TCO) برای تحلیل اقتصادی سیستم ضروری است.

۲.۴.۴ شناسایی نقاط ضعف

در غیاب مستندات کامل، مهندسی معکوس برای درک منطق و بازیابی طراحی به‌کار می‌رود. ابزارهای هوش مصنوعی مانند Copilot GitHub می‌توانند وابستگی‌ها، فراخوانی‌ها و ساختارهای منطقی را استخراج کرده و مستندات به‌روز تولید کنند.

۳.۴.۴ طراحی مجدد معماری و پیاده‌سازی

بازطراحی معماری معمولاً شامل گذار از ساختارهای یکپارچه به معماری‌های توزیع‌شده مانند میکروسرویس‌ها است. چالش‌های کلیدی این گذار عبارتند از:

- افزایش هزینه‌های زیرساختی و تست برای هر سرویس جدید.
- از دست رفتن تضمین‌های ACID و نیاز به مدیریت ثبات نهایی (Eventual Consistency).
- استفاده از الگوهایی مانند ساگا (Saga) و تضمین هم‌توانایی (Idempotency) برای هماهنگی تراکنش‌ها.

۴.۴.۴ استراتژی‌های مهاجرت

انتخاب روش مهاجرت بستگی به میزان تحمل ریسک سازمان دارد:

- **مهاجرت انفجار بزرگ (Big Bang Migration):** سوئیچ فوری از سیستم قدیمی به سیستم جدید؛ پرریسک و مناسب سیستم‌های غیر بحرانی.
- **مهاجرت افزایشی (Incremental Migration) یا الگوی انجیر خفه‌کننده:** جایگزینی تدریجی بخش‌ها و اجرای همزمان سیستم‌های قدیم و جدید برای کاهش ریسک.

فصل ۵

چرایی نیاز به مهندسی معکوس در تکامل نرم‌افزار

۱.۵ مقدمه و تعریف مهندسی معکوس

مهندسی معکوس (Reverse Engineering) در مهندسی نرم‌افزار به فرایندی گفته می‌شود که در آن یک نرم‌افزار موجود مورد تحلیل دقیق قرار می‌گیرد تا ساختار درونی، اجزا، وابستگی‌ها و منطق عملکرد آن شناخته شود، بدون این‌که لزوماً تغییری در سیستم ایجاد گردد [؟].

هدف اصلی مهندسی معکوس، بازیابی دانش از دست‌رفته یا مستندسازی‌نشده درباره‌ی سیستم است. به کمک مهندسی معکوس می‌توان فهمید که نرم‌افزار چگونه طراحی شده، اطلاعات چگونه در آن جریان دارد و بخش‌های مختلف آن چه ارتباطی با هم دارند.

برای مثال، اگر نرم‌افزاری در دسترس باشد ولی مستندات طراحی آن موجود نباشد، با مهندسی معکوس می‌توان از روی کدها و فایل‌های اجرایی، مستندات و مدل‌های طراحی را بازسازی کرد. این کار در پروژه‌هایی که نرم‌افزارهای قدیمی (Legacy Systems) مورد استفاده قرار می‌گیرند، بسیار اهمیت دارد [؟].

۱.۱.۵ تمایز آن با Reengineering و Refactoring

مفاهیم مهندسی معکوس، بازمهندسی (Reengineering) و بازآرایی کد (Refactoring) هرچند مشابه‌اند، ولی اهداف متفاوتی دارند.

- **مهندسی معکوس (Reverse Engineering):** هدف آن درک سیستم موجود است؛ یعنی بررسی و تحلیل بدون تغییر کد منبع.
- **بازمهندسی (Reengineering):** پس از شناخت کامل سیستم، آن را بازطراحی یا بازنویسی می‌کنیم تا عملکرد بهتر یا نگهداری آسان‌تری داشته باشد.
- **بازآرایی کد (Refactoring):** تمرکز بر بهبود ساختار درونی کد منبع است، بدون اینکه رفتار کلی نرم‌افزار تغییر کند.

می‌توان گفت:

مهندسی معکوس شناخت سیستم
بازمهندسی شناخت + تغییر ساختار کلی
بازآرایی اصلاح درونی کد بدون تغییر عملکرد

در چرخه‌ی عمر نرم‌افزار، مهندسی معکوس نقش مهمی در مرحله‌ی نگهداری (Maintenance) دارد، زیرا در این مرحله معمولاً نیاز به درک مجدد از ساختار و منطق سیستم احساس می‌شود.

۲.۵ دلایل نیاز به مهندسی معکوس

۱.۲.۵ فقدان مستندات یا مستندات ناقص

بسیاری از سیستم‌های نرم‌افزاری بدون مستندات کافی توسعه یافته‌اند یا مستندات آن‌ها در طول زمان از بین رفته است [؟]. در این شرایط، مهندسی معکوس به تیم توسعه کمک می‌کند تا از روی نرم‌افزار، مستندات طراحی و نمودارهای سیستم را بازسازی کند.

۲.۲.۵ تحلیل سیستم‌های قدیمی (Legacy Systems)

در سازمان‌ها هنوز از سیستم‌هایی استفاده می‌شود که بر پایه فناوری‌های قدیمی ساخته شده‌اند. مهندسی معکوس به توسعه‌دهندگان کمک می‌کند تا ساختار کلی این سیستم‌ها را درک کنند و در صورت نیاز آن‌ها را به فناوری‌های جدید منتقل نمایند.

۳.۲.۵ درک ساختار و منطق سیستم‌های موجود

گاهی نرم‌افزار توسط تیم‌های مختلف توسعه یافته و در نتیجه کدها پیچیده و نامنظم شده‌اند. مهندسی معکوس ابزاری برای درک ارتباط بین ماژول‌ها، کلاس‌ها و داده‌ها فراهم می‌کند و درک درستی از منطق سیستم به تیم توسعه می‌دهد.

۴.۲.۵ تسهیل مهاجرت به فناوری‌های جدید

تغییر پلتفرم‌ها و ابزارها اجتناب‌ناپذیر است. برای مثال، ممکن است سازمانی بخواهد نرم‌افزار خود را از نسخه‌ی دسکتاپ به تحت وب منتقل کند. مهندسی معکوس امکان تحلیل دقیق سیستم فعلی را فراهم می‌کند تا مهاجرت بدون خطا و از دست دادن داده انجام گیرد [؟].

۳.۵ چالش‌ها و محدودیت‌ها

در این فصل، به بررسی چالش‌ها و محدودیت‌های موجود در به‌کارگیری رویکرد DevOps در کنار مهندسی معکوس نرم‌افزار پرداخته می‌شود. هدف از این بخش، شناسایی عواملی است که می‌توانند بر اثربخشی، اعتبار و قابلیت تعمیم نتایج حاصل از اجرای این رویکردها تاثیرگذار باشند. محدودیت‌های شناسایی‌شده در چهار محور اصلی شامل مسائل حقوقی و مالکیت فکری، هزینه و زمان‌بر بودن فرآیند، تفسیر نادرست منطق کد، و ریسک‌های امنیتی مورد تحلیل قرار گرفته‌اند. هر یک از این عوامل، به‌صورت مستقیم یا غیرمستقیم می‌توانند مانعی در مسیر پیاده‌سازی مؤثر DevOps و بازمهندسی سیستم‌های نرم‌افزاری در محیط‌های واقعی ایجاد کنند و ضرورت به‌کارگیری رویکردی میان‌رشته‌ای و تصمیم‌گیری دقیق در این زمینه را نشان می‌دهند.

مشکلات حقوقی و مالکیت فکری

مهندسی معکوس نرم‌افزار معمولاً با محدودیت‌های قانونی و حقوقی گسترده‌ای همراه است. بسیاری از سیستم‌های نرم‌افزاری موجود در شرکت‌های بزرگ، تحت مجوزهای اختصاصی، قراردادهای توسعه یا توافق‌نامه‌های عدم افشا (NDA) طراحی و نگهداری می‌شوند. در چنین شرایطی، هرگونه تلاش برای تحلیل معکوس، استخراج کد منبع یا بازطراحی اجزای نرم‌افزار می‌تواند نقض حق مالکیت فکری محسوب شود [۲]. به‌ویژه در کشورهایی با نظام حقوقی سختگیرانه مانند ایالات متحده، قوانین

کپی‌رایت و پتنت می‌توانند مانع هرگونه مهندسی معکوس حتی با هدف بهبود سازگاری یا پایداری سیستم شوند [۱۶]. از سوی دیگر، محدودیت‌های بین‌المللی نیز موجب پیچیدگی بیشتر می‌شوند. در محیط‌های چندملیتی، تعریف و اجرای حقوق مالکیت نرم‌افزار می‌تواند میان کشورها متفاوت باشد و در نتیجه، دسترسی به کد یا داده‌های واقعی جهت تحلیل علمی محدود گردد [۱۴]. همچنین، تضاد میان نوآوری و حفاظت از مالکیت فکری چالشی فلسفی در این حوزه ایجاد کرده است. برخی محققان معتقدند که قوانین سختگیرانه‌ی IP، پیشرفت فناوری را کند می‌کنند زیرا مانع از یادگیری از سیستم‌های پیشین می‌شوند [۷]. در مقابل، گروهی دیگر بر این باورند که مهندسی معکوس بی‌رویه بدون رعایت مجوزها، ریسک سرقت فناوری و نقض حقوق مولف را افزایش می‌دهد. در این تحقیق نیز، به دلیل عدم دسترسی به داده‌های واقعی شرکت‌ها و محدودیت حقوقی تحلیل نمونه‌های صنعتی، بررسی‌های تجربی محدود به جنبه‌های نظری باقی مانده است.

هزینه و زمان‌بر بودن

از منظر اقتصادی و اجرایی، بازمهندسی نرم‌افزار فرآیندی پرهزینه و زمان‌بر است که نیازمند منابع انسانی، زیرساختی و مالی قابل‌توجهی است [۱۵]. در گزارش‌های صنعتی آمده است که شرکت‌ها سالانه میلیاردها دلار صرف نگهداری و بازطراحی سیستم‌های قدیمی خود می‌کنند و در بسیاری از موارد، هزینه‌ی بازمهندسی از هزینه‌ی توسعه‌ی مجدد سیستم جدید نیز بیشتر است [۱۷]. عوامل متعددی در افزایش این هزینه موثرند. برای نمونه، نبود مستندات کافی، نیاز به تحلیل وابستگی‌ها، بازسازی مدل داده‌ها، باز طراحی معماری و آزمون‌های مکرر همه باعث افزایش زمان اجرای پروژه می‌شوند. هرچه ساختار کد قدیمی‌تر و پیچیده‌تر باشد، زمان تحلیل و اصلاح آن به‌صورت تصاعدی افزایش می‌یابد [۱۱]. به‌علاوه، یکی از مشکلات رایج در پروژه‌های بازمهندسی، برآورد نادرست هزینه و زمان است. بسیاری از سازمان‌ها در آغاز پروژه تخمین دقیقی از میزان بدهی فنی و سطح ناسازگاری فناوری ندارند؛ در نتیجه، با افزایش غیرمنتظره‌ی هزینه‌ها، پروژه در میانه‌ی راه متوقف یا محدود می‌شود [۳].

تفسیر نادرست از منطق کد

در فرآیند بازمهندسی، درک صحیح از منطق درونی نرم‌افزار اهمیت حیاتی دارد. با این حال، بسیاری از سیستم‌های میراثی فاقد مستندات کامل هستند و مهندسان مجبورند رفتار سیستم را تنها از طریق تحلیل کد استنباط کنند. این امر منجر به تفسیر نادرست از منطق برنامه و روابط میان اجزای آن می‌شود [۹]. مطالعات نشان می‌دهند که درصد قابل‌توجهی از خطاهای به‌وجودآمده پس از بازمهندسی، ناشی از برداشت اشتباه از منطق کسب‌وکار و وابستگی‌های داخلی سیستم است [۱۰]. علاوه بر این، خروج

نیروهای کلیدی از سازمان و از بین رفتن دانش ضمنی باعث می‌شود که درک دقیق از چرایی و چگونگی تصمیمات گذشته از بین برود [۸]. ابزارهای خودکار تحلیل معنایی و مدل‌سازی معکوس هنوز در بسیاری از محیط‌های صنعتی به‌طور کامل توسعه نیافته‌اند، و این امر احتمال سوءبرداشت از منطق کد را بیشتر می‌کند.

ریسک‌های امنیتی

ریسک‌های امنیتی از مهم‌ترین موانع در مسیر بازمهندسی و بازطراحی نرم‌افزار محسوب می‌شوند. بسیاری از سیستم‌های قدیمی بر پایه‌ی فناوری‌ها و چارچوب‌هایی بنا شده‌اند که دیگر به‌روزرسانی نمی‌شوند [۶]. این وضعیت باعث می‌شود که آسیب‌پذیری‌های شناخته‌شده برای مدت طولانی در سیستم باقی بمانند و مهاجمان بتوانند از آن‌ها سوءاستفاده کنند [۵]. در فرآیند بازمهندسی، این خطر وجود دارد که مهاجرت داده‌ها، تقسیم ماژول‌ها یا اتصال سیستم‌های جدید با سیستم‌های قدیمی، مسیرهای جدیدی برای نفوذ ایجاد کند. همچنین، اگر فرآیند DevOps به‌درستی با الزامات امنیتی یکپارچه نشود، اتوماسیون نادرست می‌تواند دروازه‌هایی برای نفوذ به محیط تولید باز کند [۱]. یکی دیگر از چالش‌های امنیتی، ضعف در مدیریت وصله‌های امنیتی است. پژوهش Dissanayake و همکاران [۴] نشان می‌دهد که کمتر از ۲۰ درصد از سازمان‌ها فرآیند وصله‌گذاری خود را به‌صورت نظام‌مند و خودکار انجام می‌دهند، که این امر احتمال بروز آسیب‌پذیری در چرخه‌ی بازمهندسی را افزایش می‌دهد.

۴.۵ مطالعه موردی (Case Study)

مثال از تحلیل معکوس یک سیستم قدیمی یا نرم‌افزار متن‌باز

یکی از مطالعات شاخص در این حوزه، پژوهش Reverse engineering a legacy software in a complex system: A systems engineering approach است که توسط Maximiliano Moraga و Yang-Yang Zhao در سال ۲۰۱۸ منتشر شده است؛ در این تحقیق یک نرم‌افزار میراثی که در قالب بخشی از سیستم پیچیده‌ای قرار داشت، مورد تحلیل معکوس قرار گرفت تا دلیل شکل‌گیری ساختار، منطق عملکرد، و جایگاه آن در بستر کلی سیستم بازشناسی شود [۱۲]. در این مطالعه، تیم محققان با استفاده از مدل CAFCR (Customer Objectives – Application – Function – Component – Resources) و ابزارهای مهندسی معکوس، توانستند نقشه راه مرحله‌ای برای ارتقای تدریجی و همزمان نگه‌داری و توسعه نرم‌افزار میراثی تدوین کنند [۱۲]. به‌عنوان مثال، ابتدا نمودارهای رابطه‌ای بین

مؤلفه‌ها، وابستگی‌های زمان‌بر و حرکت از معماری مونوپولی به معماری ماژولار استخراج شد، سپس بر اساس آن تصمیماتی برای بهبود کارایی، ارتقای قابلیت نگهداری و افزون‌کردن کارکردهای جدید اتخاذ گردید [۱۲]. مطالعه دیگری تحت عنوان Case Studies in Model-Driven Reverse Engineering توسط A. Pascal و همکاران در سال ۲۰۱۹ ارائه شده است که در آن بازمهندسی سه نرم‌افزار عملیاتی با استفاده از رویکرد مدل‌محور بررسی شده است؛ در این نمونه، استخراج مدل‌های سطح بالا، تحلیل ماژول‌ها و طراحی مجدد با هدف کاهش فرسایش معماری انجام شده است [۱۳]. این مثال‌ها نشان می‌دهند که تحلیل معکوس در نرم‌افزارهای میراثی صرفاً جهت استخراج کد نیست، بلکه درک منطق کسب‌وکار، وابستگی‌های پنهان، و ساختار معماری را نیز امکان‌پذیر می‌کند؛ ولی همزمان باید توجه داشت که هر پروژه پژوهشی یا صنعتی با محدودیت‌ها و پیچیدگی‌های خاص خود مواجه است.

بررسی خروجی‌های حاصل از فرآیند مهندسی معکوس

در پروژه Moraga Zhao، خروجی‌های مهمی حاصل شده است: از جمله بازسازی نمودار زمینه (context diagram) برای نرم‌افزار مورد بررسی، استخراج اهداف مشتریان، مشخص شدن معیارهای کیفیت در رابطه با بازار هدف و ترکیب آن با وابستگی فنی مؤلفه‌ها، که منجر به تدوین نقشه‌راه برای بازمهندسی تدریجی نرم‌افزار شد [۱۲]. این نقشه راه به شرکت امکان داد تا ضمن ادامه نگهداری سیستم قدیمی، به‌تدریج عملکردهای جدید را نیز افزوده و قابلیت نگهداری را ارتقا دهد. در مطالعه Pascal، یکی دیگر از خروجی‌های کلیدی، استخراج مدل‌های معماری (architecture models) و فرم‌های بصری وابستگی‌ها میان ماژول‌ها بود؛ این مدل‌ها کمک کردند تا مسیرهای پرتکرار تغییرات، نقاط بحرانی در سیستم و اجزایی که بیشترین پیچیدگی را داشتند شناسایی شوند [۱۳]. همچنین گزارش شده که این مدل‌سازی موجب کاهش هزینه نگهداری و کاهش فرسایش معماری شده است. علاوه بر این، خروجی‌های عملی دیگری نیز شامل مستندسازی بازگشتی (redocumentation) سیستم، بازیابی دانش ضمنی کارکنان قدیمی، انتقال آن به اعضای تیم جدید، کاهش وابستگی به افراد خاص، و آماده‌سازی سیستم برای استقرار روش‌های نوین مانند DevOps بود. به‌عنوان مثال، مکانیسم‌های اتوماسیون استقرار (CI/CD) و کانتینری‌سازی پس از مهندسی معکوس بهتر قابل پیاده‌سازی شدند زیرا ساختار ماژولار بهتر درک شده بود. با این حال، باید به این نکته نیز توجه شود که خروجی‌های مهندسی معکوس معمولاً به‌صورت کامل قابل تعمیم نیستند؛ یعنی مدل‌ها، نقشه‌ها و تصمیماتی که در یک سازمان حاصل شده، ممکن است در سازمان دیگر با زیرساختی متفاوت، قابل اجرا یا مؤثر نباشند. همچنین کیفیت خروجی‌ها وابسته به میزان دسترسی به کد، مستندسازی قبلی، همکاری تیم‌های پیشین، و ابزارهای تحلیل مورد استفاده است؛ در محیط‌هایی که مستندات کم است، خطای استخراج منطق می‌تواند زیاد شود.

کتاب‌نامه

[۱] Atiba. Vulnerabilities in using legacy software: Key risks and mitigations، ۲۰۲۵.

[۲] National Research Council. Intellectual Property Issues in Software. National Academies Press، Washington، D.C.، ۱۹۹۱.

[۳] DevSquad. ۷ costs of maintaining legacy systems & how to avoid them، ۲۰۲۵.

[۴] N. Dissanayake، A. Jayatilaka، M. Zahedi و M. Ali Babar. Software security patch management — a systematic literature review of challenges، approaches، tools and practices. preprint، arXiv، ۲۰۲۰.

[۵] HeroDevs. How outdated systems and legacy software are fueling modern cyber attacks، ۲۰۲۴.

[۶] Integrity۳۶۰. How legacy software and hardware is a ticking cyber security risk time-bomb، ۲۰۲۳.

[۷] Watchdog. IP Reverse engineering law: Understanding restrictions and how to minimize risks، ۲۰۲۱.

[۸] JoIV Journal. Overview of software re-engineering concepts، models and ap- proaches، ۲۰۲۳.

[۹] Medium. When code becomes the product: Lessons in technical debt and missed opportunities، ۲۰۲۳.

[۱۰] ModelCode AI Blog. The biggest risks of misinterpreting business logic during legacy modernization، ۲۰۲۴.

[۱۱] ModLogix. software Legacy re-engineering: and Risks mitigations، ۲۰۲۲.

[۱۲] Zhao. Y. and Moraga M. Reverse engineering a software legacy in a complex system: A engineering systems approach. In INCOSE International Symposium، volume ۲۸، pages ۱۲۵۰–۱۲۶۴، ۲۰۱۸.

[۱۳] Pascal A. et al. Case studies in model-driven reverse engineering. In Proceed- ings of the ۱۱th Joint International Conference on Knowledge Discovery، Knowledge Management and Knowledge Engineering (IC۳K)، volume ۲، pages ۷۳۱–۷۴۰، ۲۰۱۹.

[۱۴] Quarkslab. Reverse engineering: A threat to intellectual property of innovations؟، ۲۰۲۳.

[۱۵] RecordPoint. The hidden costs of maintaining legacy systems، ۲۰۲۴.

[۱۶] TTC Consultants. Reverse engineering and intellectual property rights: Balancing innovation، legal considerations، ۲۰۲۴.

[۱۷] vFunction. How much does it cost to maintain software legacy systems؟، ۲۰۲۲.

[۱۸] نویسنده اول and نویسنده دوم. عنوان مقاله نمونه. نام مجله، ۱۰(۲):۱۲۳–۱۴۵، ۱۴۰۲.