دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

عنوان گزارش فارسی

زیرعنوان گزارش

نویسندگان:

محمدسینا اله‌کرم	**محمد اکبرپورجنت**	**محمد شمس الدینی**	**رضا لشنی زند**
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

۱۰ 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

۱.۱ مقدمه‌ای بر مهندسی نرم‌افزار

مهندسی نرم‌افزار شاخه‌ای از مهندسی است که به مطالعه، طراحی، توسعه، آزمون و نگهداری سیستم‌های نرم‌افزاری می‌پردازد. هدف اصلی آن، ایجاد نرم‌افزارهایی با کیفیت بالا، قابل اعتماد، کارایی مناسب، مقرون به صرفه و نگهداری آسان است. برخلاف برنامه‌نویسی صرف، مهندسی نرم‌افزار بر اصول علمی، متدولوژی‌های ساختاریافته، و ابزارهای مهندسی برای مدیریت پیچیدگی پروژه‌های نرم‌افزاری بزرگ تمرکز دارد. با رشد سریع فناوری اطلاعات و افزایش نیاز به سیستم‌های نرم‌افزاری در حوزه‌های مختلف مانند بانکداری، آموزش، بهداشت و صنعت، مهندسی نرم‌افزار به یکی از حیاتی‌ترین رشته‌های فناوری تبدیل شده است. این علم تلاش می‌کند تا توسعه نرم‌افزار را از یک فعالیت هنری یا تجربی به یک فرآیند نظام‌مند و قابل تکرار تبدیل کند.[۱]

۲.۱ تاریخچه‌ی فرایندهای توسعه نرم‌افزار

۱.۲.۱ مدل کد و فیکس (Code-and-Fix)

در دهه‌های ۱۹۵۰ و ۱۹۶۰، توسعه نرم‌افزار عمدتاً به‌صورت "کد و فیکس" انجام می‌شد. در این روش، تیم توسعه مستقیماً شروع به نوشتن کد می‌کرد و در صورت بروز خطا یا مشکل، آن را در حین کار

اصلاح می‌نمود. هیچ مستندسازی، برنامه‌ریزی دقیق یا تحلیل اولیه وجود نداشت.[۱]

مزایا: سرعت شروع بالا و مناسب برای پروژه‌های کوچک و کوتاه‌مدت.[۱]

معایب: نگهداری دشوار، افزایش هزینه در مراحل پایانی، نبود امکان پیش‌بینی خطاها و زمان تحویل.[۱]

۲.۲.۱ مدل آبشاری (Waterfall)

مدل آبشاری در دهه‌ی ۱۹۷۰ معرفی شد و نخستین مدل ساختارمند مهندسی نرم‌افزار به‌شمار می‌رود. این مدل شامل مراحلی پی‌درپی است که هر مرحله پس از اتمام مرحله‌ی قبل آغاز می‌شود. مراحل اصلی آن عبارت‌اند از: تحلیل نیازمندی‌ها، طراحی سیستم، پیاده‌سازی، آزمون، استقرار و نگهداری. در این روش، هر مرحله باید به‌طور کامل قبل از شروع مرحله بعدی به پایان برسد.[۱]

مزایا: ساختار مشخص و ساده، مستندسازی کامل و مناسب برای پروژه‌های با نیازهای پایدار.[۱]

معایب: انعطاف‌پذیری پایین در مواجهه با تغییرات، عدم امکان بازگشت به مراحل قبلی، تاخیر در کشف خطاها تا مراحل پایانی و سختی در تعامل مداوم با مشتری. اغلب برای مشتری مشکل است که تمامی نیازهای خود را به طور کامل و مشخص بیان کند، مدل آبشاری به این امر نیاز داشته و در مواجه با عدم قطعیت طبیعی که در آغاز بسیاری از پروژه‌ها وجود دارد، مشکل دارد.[۱]

با وجود محدودیت‌ها، مدل آبشاری هنوز در پروژه‌های دولتی و نظامی با الزامات دقیق مورد استفاده قرار می‌گیرد.[۱]

۳.۲.۱ مدل افزایشی و تکاملی (Incremental & Evolutionary)

در دهه ۱۹۸۰، با رشد نیاز به سیستم‌های پویا و قابل انطباق، مدل‌های افزایشی و تکاملی ظهور کردند. مدل افزایشی، مدل آبشاری را به طور تکرار شونده به کار میگیرد. در این مدل، نرم‌افزار در چند نسخه یا "افزونه" تولید می‌شود و هر نسخه بخشی از قابلیت‌های سیستم نهایی را ارائه می‌دهد. مدل تکاملی نیز بر پایه بازخورد مداوم از کاربران و بهبود تدریجی نسخه‌ها بنا شده است.[۱]

مزایا: تحویل سریع نسخه‌های اولیه، امکان دریافت بازخورد از کاربر، امکان اعمال تغییرات در طول توسعه و کاهش ریسک پروژه.[۱]

معایب: نیاز به برنامه‌ریزی دقیق و هماهنگی بین نسخه‌ها، و گاهی پیچیدگی در مدیریت تغییرات.[۱]

این رویکرد زمینه‌ساز مدل‌های مدرن‌تر مانند مدل مارپیچی و روش‌های چابک شد.

۴.۲.۱ مدل مارپیچی (Spiral Model)

مدل مارپیچی که توسط بَری بوم در سال ۱۹۸۶ معرفی شد، ترکیبی از مدل آبشاری و تکاملی است و بر تحلیل ریسک در هر تکرار تمرکز دارد. این مدل شامل چهار فاز تکرارشونده است: برنامه‌ریزی، تحلیل ریسک، مهندسی و ارزیابی. پروژه در چندین چرخه (مارپیچ) تکرار می‌شود تا محصول نهایی به بلوغ برسد.[۱]

با شروع فرآیند، تیم مهندسی نرم‌افزار در جهت عقربه‌های ساعت، حرکت در مارپیچ را آغاز می‌کند و این کار از مرکز شروع می‌شود. اولین مدار حول مارپیچ ممکن است منجر به تولید مشخصه محصول شود. با عبور از هر مرحله منطقه برنامه‌ریزی، کارهای تطابقی با طرح پروژه صورت می‌گیرد. هزینه و زمانبندی بر اساس بازخورد ارزیابی مشتری، تنظیم می گردند. علاوه بر آن مدیر پروژه تعداد تکرارهای تنظیم شده لازم برای تکمیل نرم افزار را تعیین میکند.[۱]

مزایا: مدیریت مؤثر ریسک‌ها، انعطاف‌پذیری بالا، مناسب برای پروژه‌های بزرگ و پیچیده.[۱]

معایب: نیاز به تخصص بالا در تحلیل ریسک و افزایش هزینه نسبت به مدل‌های ساده‌تر.[۱]

۵.۲.۱ مدل چابک (Agile) و ظهور DevOps

در دهه ۲۰۰۰، با انتشار مانیفست چابک (Agile Manifesto)، پارادایم جدیدی در مهندسی نرم‌افزار شکل گرفت. روش‌های چابک مانند XP، اسکرام (Scrum) و کانبان (Kanban) بر همکاری تیمی، تحویل سریع نسخه‌های قابل اجرا، پاسخ به تغییرات و ارتباط مستمر با مشتری تمرکز دارند.[۱]

مزایا: تعامل مستقیم با مشتری، بازخورد سریع، چرخه تحویل کوتاه‌تر، کیفیت بالاتر، افزایش رضایت مشتری، و کاهش خطاهای عملیاتی.[۱]

معایب: نیاز به فرهنگ سازمانی جدید، ابزارهای پیشرفته و یادگیری مستمر، دشواری در مستندسازی رسمی.[۱]

به مرور، DevOps به‌عنوان گامی تکمیلی در این مسیر پدیدار شد. با گسترش DevOps، چرخه‌ی توسعه و عملیات به‌صورت یکپارچه درآمد تا تحویل مداوم (Continuous Delivery)، استقرار خودکار (Continuous Deployment) و نظارت مستمر فراهم شود. DevOps به نوعی ادامه و بلوغ طبیعی Agile به‌شمار می‌رود که فاصله‌ی بین تیم توسعه و تیم زیرساخت را از میان برداشته است. [۱]

۳.۱ نقش بازخورد و تکامل در مهندسی نرم‌افزار

بازخورد نقش حیاتی در فرایند توسعه نرم‌افزار دارد. بدون دریافت بازخورد از کاربران، ذی‌نفعان یا اعضای تیم، نرم‌افزار نمی‌تواند با نیازهای واقعی محیط و کاربران هماهنگ شود. تکامل نرم‌افزار نتیجهٔ بازخوردهای پی‌درپی و اصلاحات مستمر است. در مهندسی نرم‌افزار مدرن، بازخورد از طریق آزمون‌های خودکار، بازبینی کد (Code Review)، و جلسات مرور عملکرد پروژه (Sprint Review) جمع‌آوری می‌شود. این بازخوردها موجب بهبود کیفیت، افزایش رضایت مشتری و کاهش هزینه‌های بلندمدت می‌شوند.

۴.۱ مفهوم چرخه عمر نرم‌افزار (SDLC)

چرخه عمر توسعه نرم‌افزار (Software Development Life Cycle) مجموعه‌ای از مراحل منظم برای تولید، استقرار و نگهداری نرم‌افزار است که در طول تاریخ توسعه‌ی مهندسی نرم‌افزار تکامل یافته است.[۲]

در دهه ۱۹۶۰، مدل کد و فیکس بدون ساختار مشخص به‌کار می‌رفت و مفهومی از چرخه عمر وجود نداشت. با رشد پروژه‌ها و پیچیدگی سیستم‌ها در دهه ۱۹۷۰، مدل آبشاری معرفی شد و برای نخستین‌بار مراحل SDLC به‌صورت خطی تعریف شدند: تحلیل، طراحی، پیاده‌سازی، تست و نگهداری.[۲]

در دهه ۱۹۸۰، مدل‌های افزایشی و تکاملی مفهوم تکرارپذیری را وارد SDLC کردند. نرم‌افزار در چند چرخه‌ی کوچک توسعه می‌یافت و بازخورد کاربران باعث تکامل تدریجی محصول می‌شد.[۲]

مدل مارپیچی در دهه ۱۹۹۰ با تمرکز بر مدیریت ریسک، SDLC را به فرآیندی پویا و تکرارشونده تبدیل کرد. در هر چرخه، برنامه‌ریزی، تحلیل ریسک، طراحی و ارزیابی انجام می‌شد.[۲]

در نهایت، با ظهور چابک (Agile) و سپس DevOps در دهه ۲۰۰۰ به بعد، SDLC از رویکردهای سنگین و مستندسازی محور فاصله گرفت و به فرآیندی سریع، انعطاف‌پذیر و مبتنی بر بازخورد تبدیل شد. اکنون SDLC شامل فازهای پویا و پیوسته‌ای مانند برنامه‌ریزی، توسعه، تست خودکار، استقرار و نگهداری مستمر است که به بهبود مداوم نرم‌افزار و رضایت کاربر منجر می‌شود.[۲]

فازهای اصلی SDLC

برنامه‌ریزی (Planning): در این مرحله اهداف پروژه، نیازمندی‌های کلی، منابع، بودجه و زمان‌بندی

تعیین می‌شوند. تحلیل ریسک‌ها و تهیه طرح مدیریت پروژه نیز در این فاز انجام می‌گیرد.[۲]

تحلیل نیازمندی‌ها (Requirement Analysis): تیم تحلیل، نیازهای کاربران و ذی‌نفعان را شناسایی و مستند می‌کند. خروجی این فاز، سند مشخصات نیازمندی‌های نرم‌افزار (SRS) است.[۲]

طراحی سیستم :(Design) ساختار کلی سیستم، معماری نرم‌افزار، طراحی پایگاه داده و رابط کاربری مشخص می‌شود. در این مرحله، مدل‌های UML و دیاگرام‌های مختلف برای شفاف‌سازی طراحی استفاده می‌شوند.[۲]

پیاده‌سازی :(Implementation) کدنویسی بر اساس طراحی انجام می‌شود. توسعه‌دهندگان از زبان‌ها، فریم‌ورک‌ها و ابزارهای مختلف برای تولید نرم‌افزار استفاده می‌کنند.[۲]

تست :(Testing) در این فاز، نرم‌افزار از نظر عملکردی، امنیتی، سازگاری و کارایی مورد آزمون قرار می‌گیرد. هدف، شناسایی و رفع خطاها پیش از استقرار است.[۲]

نگهداری :(Maintenance) پس از استقرار نرم‌افزار، ممکن است نیاز به اصلاح خطاها، افزودن قابلیت‌های جدید یا بهینه‌سازی عملکرد باشد. نگهداری مناسب، عمر مفید نرم‌افزار را افزایش می‌دهد و از افت کیفیت آن جلوگیری می‌کند.[۲]

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...

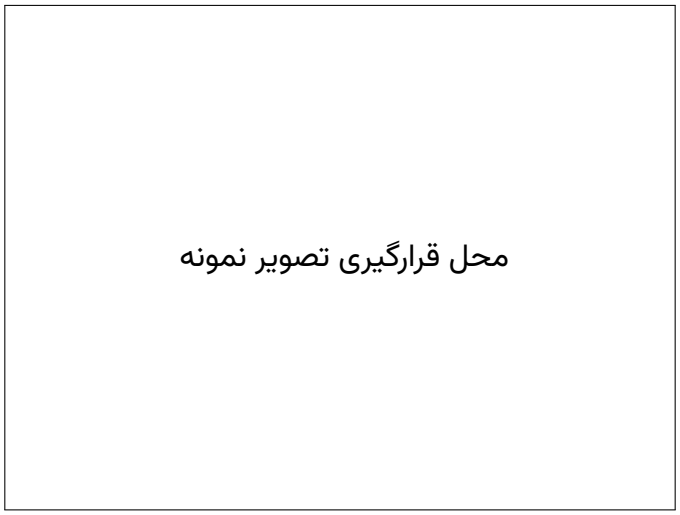
مثال ۱.۲. مثالی برای توضیح بهتر مفهوم: می‌توان فرمول ریاضی نوشت: $f(x) = x^۲ + ۲x + ۱$

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:

- [۲۱]: 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 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.

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.

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، Elastic Stack برای مدیریت و تحلیل لاگ‌ها، Prometheus برای مانیتورینگ و هشدار و Sentry برای ردیابی خطاها در لحظه استفاده می‌شوند.

۴.۳ پیشنهادات برای کارهای آینده

بر اساس نتایج و محدودیت‌های شناسایی شده، پیشنهادات زیر برای کارهای آینده ارائه می‌شود:

۱.۴.۳ بهبودهای کوتاه‌مدت

۱. بهینه‌سازی کد برای کاهش مصرف حافظه

۲. افزودن قابلیت‌های جدید به سیستم

۳. بهبود رابط کاربری

۲.۴.۳ پیشنهادات برای تحقیقات آینده

- بررسی استفاده از روش‌های یادگیری عمیق
- توسعه نسخه توزیع شده از سیستم
- ارزیابی روی مجموعه داده‌های بزرگ‌تر
- مطالعه کاربردهای جدید در حوزه‌های دیگر

۳.۴.۳ کاربردهای بالقوه

این کار می‌تواند در زمینه‌های زیر کاربرد داشته باشد:

- صنعت و تولید
- آموزش و پژوهش
- خدمات و تجارت الکترونیک

۵.۳ فرهنگ و سازمان‌دهی در DevOps

۱.۵.۳ همکاری میان تیم توسعه و عملیات

DevOps تنها مجموعه‌ای از ابزارها و فرایندهای فنی نیست، بلکه یک تغییر فرهنگی و سازمانی عمیق در نحوه‌ی همکاری میان تیم‌های توسعه (Development) و عملیات (Operations) است. این فرهنگ بر پایه‌ی اعتماد، ارتباط، شفافیت و مسئولیت مشترک بنا شده است. بر اساس پژوهش [۱۰]، DevOps پیش از آن‌که رویکردی فنی باشد، نوعی تغییر در نگرش سازمانی است که موجب نزدیکی میان تیم‌های مختلف و شکل‌گیری ذهنیت همکاری می‌شود.

در مدل‌های سنتی، توسعه‌دهندگان پس از نوشتن کد، آن را تحویل تیم عملیات می‌دادند تا در محیط واقعی مستقر شود. نتیجه‌ی این جدایی، بروز مشکلاتی مانند عدم هماهنگی، خطاهای زیاد در استقرار و تأخیر در تحویل بود. DevOps با هدف رفع این شکاف به‌وجود آمد تا توسعه و عملیات به‌صورت یک واحد عمل کنند و مسئولیت موفقیت یا شکست نرم‌افزار را به‌صورت مشترک بر عهده بگیرند.

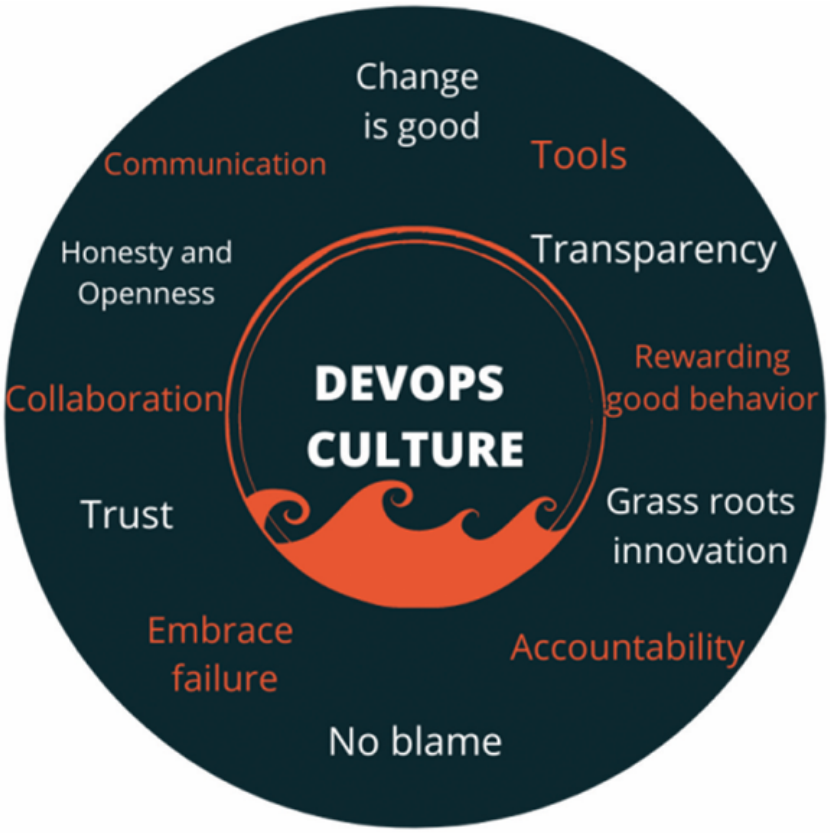
۲.۵.۳ مؤلفه‌های اصلی فرهنگ DevOps

ارتباط باز و مداوم: تیم‌ها باید به‌طور پیوسته با یکدیگر در ارتباط باشند. ابزارهایی مانند Slack یا Microsoft Teams برای گفت‌وگوهای لحظه‌ای، و Jira برای پیگیری وظایف به‌کار می‌روند. این ارتباط مداوم باعث می‌شود تصمیم‌ها سریع‌تر گرفته شوند و مشکلات پیش از تبدیل‌شدن به بحران، شناسایی و رفع شوند. نمونه‌ی عملی آن در شرکت Atlassian دیده می‌شود که توسعه‌دهندگان و مدیران سیستم وضعیت پایپ‌لاین‌های CI/CD و استقرارها را در همان کانال‌های گفت‌وگو دنبال می‌کنند.

مسئولیت مشترک (Shared Ownership): در فرهنگ DevOps دیگر مفهوم «تحویل دادن کد و رها کردن آن» وجود ندارد. توسعه‌دهندگان در موفقیت نرم‌افزار پس از استقرار نیز نقش مستقیم دارند و در مقابل، تیم عملیات هم از مراحل طراحی و تست در جریان پروژه قرار می‌گیرد. مثال شناخته‌شده، سیاست «You build it, you run it» در شرکت Amazon است که باعث می‌شود توسعه‌دهنده نسبت به پایداری و مانیتورینگ نرم‌افزار در محیط واقعی حساس‌تر باشد.

یادگیری و بهبود مستمر (Continuous Learning): پس از هر انتشار (Release)، تیم‌ها جلساتی با عنوان Postmortem برگزار می‌کنند تا شکست‌ها و موفقیت‌ها را بررسی کنند. هدف، سرزنش افراد نیست؛ بلکه یافتن علت ریشه‌ای خطا و اصلاح فرایند است. شرکت‌هایی مانند Google از گزارش‌های Blameless Postmortem استفاده می‌کنند تا بدون مقصر جلوه‌دادن افراد، فرایندها و پیکربندی‌ها را بهبود دهند.

هم‌ترازی اهداف بین تیم‌ها (Goal Alignment): در سازمان‌های سنتی، اهداف توسعه (تحویل سریع‌تر) و عملیات (پایداری بیشتر) معمولاً در تضاد هستند. DevOps با تعریف شاخص‌های عملکرد مشترک مانند MTTR و Deployment Frequency این تضاد را کاهش می‌دهد و باعث می‌شود هر دو تیم به سمت هدف مشترک، یعنی تحویل سریع ولی پایدار نرم‌افزار، حرکت کنند. همان‌طور که در [۱۰] آمده است، هم‌ترازی هدف‌ها باعث می‌شود معیارهای ارزیابی از فردمحور به تیم‌محور تغییر کند.



شکل ۱.۳: نمایی از مؤلفه‌های فرهنگ و ذهنیت DevOps بر اساس [۱۰].

۶.۳ مزایای DevOps در تکامل نرم‌افزار

مهم‌ترین مزایای به‌کارگیری DevOps در فرایند توسعه و تکامل نرم‌افزار عبارت‌اند از:

- افزایش سرعت تحویل نرم‌افزار
- بهبود پایداری و اطمینان در استقرارها
- ارتقای کیفیت محصول
- افزایش بهره‌وری و هماهنگی تیم‌ها
- توانایی پاسخ سریع به تغییرات بازار و نیازهای کاربران

همان‌طور که در [۱۰] نیز اشاره شده است، این مزایا زمانی به‌طور کامل به دست می‌آیند که فرهنگ همکاری میان تیم‌های توسعه و عملیات که در بخش ۱.۵.۳ توضیح داده شد، در سازمان نهادینه شده باشد.

افزایش سرعت تحویل نرم‌افزار

DevOps موجب می‌شود چرخه‌ی توسعه از ایده تا تحویل نهایی کوتاه‌تر شود. با خودکارسازی مراحلی مانند ساخت، تست و استقرار، تیم‌ها می‌توانند در بازه‌های زمانی بسیار کوتاه نسخه‌های جدید ارائه دهند. به‌عنوان نمونه، شرکت Amazon روزانه هزاران استقرار جدید در زیرساخت خود انجام می‌دهد. این حجم از به‌روزرسانی تنها به لطف استفاده از خطوط خودکار CI/CD ممکن است.

بهبود پایداری و اطمینان در استقرارها

در روش‌های سنتی، استقرار نرم‌افزار اغلب با اضطراب و خطا همراه بود، زیرا تغییرات به‌صورت گسترده و یک‌باره اعمال می‌شد. DevOps این مشکل را با اعمال تغییرات کوچک و مکرر حل کرده است. نمونه‌ی شناخته‌شده، تجربهٔ Etsy است که پس از خودکارسازی استقرارها، توانست بدون توقف سرویس، استقرارهای متعدد روزانه انجام دهد.

ارتقای کیفیت محصول

تست‌های خودکار و مانیتورینگ مستمر از ارکان DevOps هستند و کمک می‌کنند خطاها در مراحل ابتدایی شناسایی و اصلاح شوند. شرکت‌هایی مانند Google با تکیه بر پایش مداوم، نرخ خرابی را کاهش داده‌اند.

افزایش بهره‌وری و هماهنگی تیم‌ها

DevOps باعث می‌شود تیم‌های توسعه، عملیات، آزمون و حتی امنیت در یک چرخه‌ی واحد کار کنند و کارهای دستی و تکراری حذف شود.

پاسخ سریع به تغییرات بازار و نیازهای کاربران

در محیط‌های پویا، چرخه‌ی بازخورد سریع که در [۱۰] بر آن تأکید شده، امکان انتشار و بازگردانی سریع ویژگی‌ها را فراهم می‌کند.

۷.۳ مطالعه‌ی موردی

شرکت Netflix با میلیون‌ها کاربر در سراسر جهان، یکی از پیشگامان در به‌کارگیری رویکرد DevOps است. مقیاس بسیار بزرگ سامانه و نیاز به ارائهٔ مداوم محتوا، این شرکت را بر آن داشت تا از شیوه‌های سنتی توسعه فاصله بگیرد و معماری‌ای پویا و مبتنی بر خودکارسازی ایجاد کند. همان‌گونه که در پژوهش [۱۰] نیز تأکید شده، موفقیت در مقیاس گسترده تنها زمانی ممکن است که فرهنگ سازمانی، ابزارها و فرآیندها هم‌زمان دگرگون شوند.

چالش‌های اولیه

در سال‌های ابتدایی فعالیت، Netflix با چند چالش اساسی روبه‌رو بود:

- استقرارهای نرم‌افزاری به‌صورت دستی انجام می‌شد و احتمال خطاهای انسانی بالا بود.
- هرگونه تغییر کوچک در سیستم می‌توانست موجب اختلال در پخش محتوا شود.
- سرورها در مراکز دادهٔ داخلی نگهداری می‌شدند و مقیاس‌پذیری آن‌ها محدود بود.

این چالش‌ها سبب شدند که Netflix در سال ۲۰۰۸ تصمیم بگیرد به زیرساخت ابری مهاجرت کند و هم‌زمان فلسفهٔ DevOps را در سازمان پیاده‌سازی کند. این تصمیم، نقطهٔ عطفی در مسیر تکامل فنی و فرهنگی شرکت بود.

معماری و ابزارهای مورد استفاده

برای تحقق اصول DevOps، Netflix مجموعه‌ای از ابزارها و فرآیندهای خودکار را توسعه داد. برخی از مهم‌ترین آن‌ها عبارت‌اند از:

- **Spinnaker:** سیستم متن‌باز ویژهٔ Netflix برای خودکارسازی خط لوله‌های CI/CD. این ابزار امکان استقرار مکرر، سریع و بدون وقفهٔ سرویس‌ها را فراهم می‌کند.

- **Chaos Monkey:** ابزاری برای آزمایش پایداری سیستم از طریق ایجاد خطاهای تصادفی در سرورها؛ هدف آن ارزیابی مقاومت سامانه در برابر شکست است.
- **Atlas** و **Vector:** ابزارهای پایش و تحلیل عملکرد سرویس‌ها که داده‌ها را به‌صورت لحظه‌ای جمع‌آوری و بررسی می‌کنند.

با این زیرساخت‌ها، Netflix قادر است روزانه صدها استقرار جدید انجام دهد، بدون آن‌که کاربران هیچ‌گونه اختلالی در سرویس احساس کنند.

فرهنگ سازمانی DevOps در Netflix

مطابق با دیدگاه مطرح‌شده در [۱۰]، یکی از عوامل کلیدی موفقیت DevOps در Netflix، نهادینه‌سازی آن در فرهنگ سازمانی است. اصول فرهنگی مهم در این شرکت شامل موارد زیر است:

- **اعتماد به تیم‌ها:** هر تیم مسئول استقرار و نگهداری سرویس‌های خود است.
- **آزادی همراه با مسئولیت:** توسعه‌دهندگان در انتخاب ابزار و روش‌ها آزادی کامل دارند، اما مسئولیت عملکرد سرویس نیز با خود آنان است.
- **بازخورد سریع:** داده‌های واقعی کاربران به‌صورت لحظه‌ای تحلیل می‌شود و تصمیم‌گیری‌ها بر پایهٔ شواهد انجام می‌گیرد.

نتایج پیاده‌سازی

اجرای اصول DevOps در Netflix منجر به بهبود چشمگیر در جنبه‌های مختلف توسعه و بهره‌برداری از سامانه شده است:

- کاهش محسوس خطاهای استقرار،
- افزایش سرعت ارائهٔ قابلیت‌های جدید،
- مقیاس‌پذیری بسیار بالا در پاسخ به رشد کاربران،
- ارتقای تجربهٔ کاربری و کاهش زمان قطعی سرویس.

به‌عنوان نمونه، در زمان اوج مصرف، سامانه‌های Netflix قادرند میلیون‌ها درخواست هم‌زمان را بدون افت کیفیت پاسخ دهند؛ قابلیتی که بدون زیرساخت خودکار و فرهنگ همکاری DevOps امکان‌پذیر نبود.

۸.۳ چالش‌های استقرار DevOps

هرچند DevOps در سال‌های اخیر به‌عنوان یکی از مؤثرترین رویکردها در توسعهٔ نرم‌افزار شناخته شده است، اما پیاده‌سازی موفق آن کار ساده‌ای نیست. همان‌گونه که در پژوهش [۱۰] نیز اشاره شده، سازمان‌ها در مسیر استقرار DevOps با موانع فنی و فرهنگی متعددی روبه‌رو می‌شوند که در صورت مدیریت‌نشدن صحیح، می‌توانند موجب کندی یا حتی شکست کل فرآیند شوند. در ادامه، مهم‌ترین چالش‌های پیاده‌سازی این رویکرد بررسی می‌شود.

مسائل امنیتی و حفظ اعتماد

با خودکار شدن فرآیندها و افزایش سرعت استقرار، امنیت به یکی از دغدغه‌های اصلی در محیط‌های DevOps تبدیل شده است. در روش‌های سنتی، بررسی‌های امنیتی معمولاً در انتهای چرخهٔ توسعه انجام می‌شد، اما در DevOps انتشارهای سریع و مکرر ممکن است سبب نادیده‌گرفتن برخی کنترل‌های حیاتی شود. برای نمونه، زمانی که تیم توسعه به‌صورت روزانه کد جدید را با شاخهٔ اصلی ادغام می‌کند، یک آسیب‌پذیری کوچک می‌تواند بلافاصله وارد محیط تولید شود. برای رفع این مشکل، رویکرد DevSecOps پیشنهاد می‌شود که در آن، امنیت از مراحل اولیهٔ توسعه در چرخهٔ عمر نرم‌افزار ادغام می‌شود. همچنین کنترل دسترسی، مدیریت کلیدها و محافظت از داده‌های حساس از مسئولیت‌های مهمی هستند که نیاز به نظارت مداوم دارند.

پیچیدگی زیرساخت و وابستگی به ابزارها

یکی دیگر از چالش‌های جدی، افزایش پیچیدگی فنی در اثر استفاده از ابزارهای متنوع است. سازمان‌ها برای پیاده‌سازی DevOps اغلب از ترکیب ابزارهایی چون Docker، Kubernetes، Jenkins و Terraform استفاده می‌کنند. هرچند این ابزارها قدرت و انعطاف بالایی دارند، اما برای تیم‌هایی که تجربهٔ کافی ندارند، می‌توانند موجب سردرگمی و کاهش بهره‌وری شوند. مطالعهٔ [۱۰] نشان می‌دهد تمرکز بیش از حد بر ابزارها ممکن است هدف اصلی DevOps یعنی همکاری مؤثر و تحویل سریع ارزش به مشتری را

تحت‌الشعاع قرار دهد. مستندسازی دقیق، آموزش منظم و طراحی زیرساخت ساده و پایدار از مهم‌ترین راهکارهای مقابله با این چالش هستند.

مقاومت فرهنگی و تغییر در شیوهٔ کار

مهم‌ترین مانع در مسیر اجرای DevOps، چالش فرهنگی درون سازمان است. برخلاف تصور رایج، DevOps صرفاً تغییر در ابزارها نیست، بلکه تحولی در نگرش، ساختار و مسئولیت‌پذیری اعضاست. در مدل سنتی، تیم‌های توسعه و عملیات معمولاً به‌صورت مجزا عمل می‌کردند و هرکدام تنها بخشی از مسئولیت را بر عهده داشتند؛ اما در DevOps مرزها از میان برداشته می‌شوند و موفقیت کل محصول، مسئولیتی جمعی است. در بسیاری از سازمان‌ها، این تغییر ذهنیت با مقاومت مواجه می‌شود — به‌ویژه در ساختارهای سلسله‌مراتبی که عادت به تفکیک نقش‌ها دارند. تجربهٔ گزارش‌شده در [۱۰] نشان می‌دهد آموزش مستمر، شفاف‌سازی اهداف و مشارکت فعال کارکنان در تصمیم‌گیری، از مؤثرترین راهکارها برای غلبه بر این مقاومت فرهنگی است.

در مجموع، استقرار موفق DevOps مستلزم آمادگی فنی و فرهنگی توأمان است. بی‌توجهی به یکی از این ابعاد می‌تواند موجب کندی در تحول سازمانی و کاهش اثربخشی کل چرخهٔ توسعه شود.

۹.۳ جمع‌بندی فصل

در این فصل نشان داده شد که DevOps فراتر از مجموعه‌ای از ابزارها یا روش‌های فنی است و در واقع یک تغییر بنیادی در فرهنگ و نگرش سازمانی به شمار می‌آید. بر اساس پژوهش [۱۰]، موفقیت در اجرای DevOps زمانی حاصل می‌شود که سازمان‌ها بر سه محور کلیدی تمرکز کنند: همکاری مستمر، مسئولیت‌پذیری مشترک و بهبود پیوسته. در چنین بستری، مرز میان تیم‌های توسعه و عملیات از میان برداشته می‌شود و کل سازمان به یک واحد منسجم در راستای تحویل ارزش به کاربر تبدیل می‌گردد.

رویکرد DevOps با اتکا به خودکارسازی، زیرساخت به‌عنوان کد (Infrastructure as Code) و چرخه‌های یکپارچهٔ CI/CD، توانسته است فاصله میان تولید نرم‌افزار و استقرار آن را به‌طور چشمگیری کاهش دهد. نتیجهٔ این تحول، تولید نرم‌افزارهایی با کیفیت بالاتر، قابلیت اطمینان بیشتر و سرعت انتشار بالاتر است. ابزارهایی مانند Jenkins، Docker و Kubernetes ستون‌های فنی این رویکرد را تشکیل می‌دهند و زمینه را برای پیاده‌سازی پایدار و مقیاس‌پذیر فرآیندها فراهم می‌کنند.

نمونه‌های موفقی همچون Netflix و Amazon نشان داده‌اند که اجرای اصول DevOps نه‌تنها موجب افزایش چابکی و مقیاس‌پذیری می‌شود، بلکه توانایی سازمان در پاسخ‌گویی به تغییرات بازار و

نیاز کاربران را نیز ارتقا می‌دهد. با این حال، همان‌گونه که در [۱۰] تأکید شده، استقرار DevOps بدون آمادگی فرهنگی و آموزشی کافی می‌تواند با چالش‌هایی چون پیچیدگی زیرساخت، ضعف در امنیت و مقاومت کارکنان روبه‌رو شود.

در نهایت می‌توان DevOps را پلی میان فرهنگ Agile و عملیات مدرن دانست؛ پلی که با تقویت ارتباط میان فناوری، فرآیند و فرهنگ همکاری، مسیر تحول دیجیتال را هموار می‌سازد. سازمان‌هایی که بتوانند میان این سه بُعد تعادل برقرار کنند، نه‌تنها در توسعهٔ نرم‌افزار بلکه در کل چرخهٔ عمر نوآوری و ارزش‌آفرینی خود به موفقیت پایدار دست خواهند یافت.

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 Load (Peak 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.

۵.۴ ابزارها و تکنیک‌های بازطراحی

فرآیند بازطراحی و نگهداری سیستم‌های قدیمی معمولاً با به کارگیری مجموعه‌ای از ابزارها و تکنیک‌های تخصصی انجام می‌شود...

۱.۵.۴ بازآرایی (Refactoring)

بازآرایی فرآیند بازسازی (restructuring) ساختار داخلی کد بدون تغییر رفتار خارجی آن است... [۱].

۲.۵.۴ مهندسی معکوس (Reverse Engineering)

مهندسی معکوس فرآیند استخراج طراحی، معماری و مشخصات یک سیستم از کد منبع موجود است...

۳.۵.۴ مهاجرت (Migration)

مهاجرت به فرآیند انتقال یک سیستم نرم‌افزاری از یک محیط تکنولوژیکی قدیمی به یک محیط جدیدتر... [۲].

کتاب‌نامه

[۱] IBM. (۲۰۲۳). Code Refactoring. Retrieved from https://www.ibm.com/think/topics/code-refactoring

[۲] GeeksforGeeks. (۲۰۲۳). Best Practices for Data Migration. Retrieved from https://www.geeksforgeeks.org/blogs/best-practices-for-data-migration

۶.۴ معیارهای تصمیم‌گیری برای بازطراحی

تصمیم‌گیری برای انجام فرآیند بازطراحی یک سیستم نرم‌افزاری، نیازمند سنجش و ارزیابی دقیق چندین معیار حیاتی است تا بتوان توجیه فنی و اقتصادی آن را به درستی بررسی کرد. مهم‌ترین این معیارها عبارتند از:

۱.۶.۴ هزینه (Cost)

برآورد دقیق تمامی هزینه‌های مستقیم و غیرمستقیم پروژه بازطراحی امری ضروری است. این هزینه‌ها شامل دستمزد تیم توسعه، هزینه‌های مربوط به خرید یا اجاره ابزارها و زیرساخت‌های جدید، هزینه‌های آموزش پرسنل و همچنین هزینه‌های احتمالی توقف یا کاهش عملکرد سیستم در حین اجرای پروژه می‌شود. این معیار باید در مقابل هزینه‌های ادامه کار با سیستم قدیمی (مانند هزینه‌های بالای نگهداری و رفع نقص) سنجیده شود.

۲.۶.۴ زمان (Time)

تخمین مدت زمان مورد نیاز برای تکمیل فرآیند بازطراحی از اهمیت بالایی برخوردار است. یک برنامه‌ریزی واقع‌بینانه باید شامل مراحل تحلیل، طراحی، پیاده‌سازی، تست و استقرار باشد. زمان‌بندی طولانی می‌تواند منجر به منسوخ شدن فناوری‌های به کار رفته در طول اجرای پروژه شود، در حالی که زمان‌بندی بسیار فشرده نیز کیفیت نهایی را به خطر می‌اندازد.

۳.۶.۴ ریسک (Risk)

ارزیابی ریسک‌های بالقوه در موفقیت پروژه بازطراحی یک گام کلیدی است. این ریسک‌ها می‌توانند شامل پیچیدگی فنی بالای سیستم ،legacy از دست دادن مهارت‌های تخصصی مورد نیاز، بروز مشکلات غیرمنتظره در حین مهاجرت داده‌ها، و مقاومت کاربران در برابر پذیرش سیستم جدید باشد. شناسایی این ریسک‌ها و برنامه‌ریزی برای مدیریت آن‌ها شانس موفقیت پروژه را افزایش می‌دهد.

۴.۶.۴ اثر بر کیفیت (Effect on (Quality

در نهایت، باید تأثیر مثبت بازطراحی بر کیفیت محصول نهایی به وضوح تعریف و اندازه‌گیری شود. این بهبود کیفیت می‌تواند به صورت افزایش کارایی ،(Performance) افزایش قابلیت اطمینان (Relia-bility)، افزایش امنیت، بهبود قابلیت نگهداری (Maintainability) و افزایش قابلیت گسترش (Scal-ability) سیستم ظاهر شود. این معیار نهایی، توجیه اصلی برای سرمایه‌گذاری روی پروژه بازطراحی محسوب می‌شود.

۷.۴ مطالعه موردی

بر اساس نتایج جستجو، اطلاعات مربوط به بازطراحی پی‌پال بیشتر بر به‌روزرسانی تجربه کاربری... [۱].

۱.۷.۴ بازطراحی اپلیکیشن PayPal

هدف اصلی از بازطراحی پی‌پال، تبدیل آن از یک ابزار ساده برای انتقال پول به یک "راهنمای سلامتی مالی"...

۲.۷.۴ بازطراحی بانکداری برای کسب‌وکارهای کوچک (فوربیکس)

در ایران، نمونه بارز بازطراحی در سیستم بانکی، ظهور "نئوبانک‌های کسب‌وکاری" مانند فوربیکس است...

کتاب‌نامه

[۱] Elastic Path. (۲۰۲۳). Deconstructing PayPal’s Radical Re-Redesign. Checkout retrieved from https://www.elasticpath.com/blog/deconstructing-paypals-radical-checkout-redesign

[۲] Think Company. (۲۰۲۳). ۶ Steps for a Successful Digital Redesign Project. Retrieved from https://www.thinkcompany.com/blog/۶-steps-for-a-successful-digital-redesign-project

۸.۴ نتیجه‌گیری نهایی و توصیه‌ها برای تیم‌های توسعه

بازطراحی نرم‌افزار یک سرمایه‌گذاری استراتژیک برای حفظ سلامت، کارایی و طول عمر سیستم‌های نرم‌افزاری محسوب می‌شود. همان‌طور که در بخش‌های پیشین بررسی شد، این فرآیند با استفاده از تکنیک‌هایی مانند بازآرایی، مهندسی معکوس و مهاجرت، و با در نظرگیری معیارهای حیاتی چون هزینه، زمان، ریسک و اثر بر کیفیت انجام می‌پذیرد. بررسی سیستم‌هایی مانند پی‌پال نیز نشان می‌دهد که یک بازطراحی موفق می‌تواند منجر به افزایش رضایت کاربر، بهبود قابلیت نگهداری و کسب مزیت رقابتی پایدار شود. در پایان، موارد کلیدی زیر می‌تواند راهنمای تیم‌های توسعه در این مسیر باشد:

۱.۸.۴ ارزیابی واقع‌بینانه و مبتنی بر داده

پیش از هر اقدامی، با استفاده از معیارهای کمّی (مانند اندازه پیچیدگی کد، تعداد باگ‌ها، هزینه نگهداری) و کیفی (رضایت کاربران و تیم توسعه) به ارزیابی دقیق نیازمندی‌های سیستم موجود بپردازید. این ارزیابی، مبنای علمی و متقاعدکننده‌ای برای تصمیم‌گیری در مورد لزوم و دامنه بازطراحی فراهم می‌کند.

۲.۸.۴ اولویت‌بندی و رویکرد تدریجی

بازطراحی کامل یک سیستم بزرگ در یک بازه زمانی کوتاه، ریسک بسیار بالایی دارد. توصیه می‌شود پروژه به بخش‌های کوچک‌تر و مستقل تقسیم شده و به صورت تدریجی و با اولویت‌بندی بر اساس ماژول‌هایی که بیشترین مشکل را ایجاد می‌کنند، اجرا شود. این رویکرد، مدیریت پروژه را آسان‌تر کرده و امکان دریافت بازخورد سریع را فراهم می‌کند.

۳.۸.۴ سرمایه‌گذاری بر روی اتوماسیون

استقرار یک خط لوله قوی یکپارچه‌سازی و تحویل مستمر (CI/CD) و یک مجموعه جامع از آزمون‌های خودکار را در اولویت قرار دهید. این امر با اطمینان از اینکه تغییرات کد، عملکرد موجود را خراب نمی‌کند، ایمنی و سرعت فرآیند بازطراحی را به طور چشمگیری افزایش می‌دهد.

۴.۸.۴ مستندسازی همگام با توسعه

فرآیند بازطراحی را فرصتی برای جبران کمبود مستندات سیستم قدیمی بدانید. همگام با پیاده‌سازی کد جدید، مستندات طراحی، معماری و نحوه راه‌اندازی را به روز کنید. این کار نگهداری سیستم را در آینده بسیار ساده‌تر خواهد کرد.

۵.۸.۴ در نظر گرفتن پیامدهای فرهنگی

بازطراحی تنها یک چالش فنی نیست، بلکه یک تغییر سازمانی است. تیم را از مزایای بلندمدت این کار آگاه سازید و برای پذیرش این تغییر و یادگیری فناوری‌ها یا روش‌های جدید، فرهنگسازی و آموزش لازم را فراهم کنید. موفقیت نهایی در گرو همراهی و مهارت تیم توسعه است.

در نهایت، بازطراحی را نه به عنوان یک هزینه، بلکه به عنوان یک ضرورت برای بقا و رشد نرم‌افزار در نظر بگیرید. یک برنامه‌ریزی دقیق، اجرای گام‌به‌گام و تمرکز بر کیفیت، می‌تواند عمر سیستم شما را طولانی کرده و ارزش آن را در بلندمدت به میزان قابل توجهی افزایش دهد.

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 [۴]. 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 [۱۹]. 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 [۱۷]. 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 [۹]. 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 [۱۸]. 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 [۲۰]. 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 [۱۴]. 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 [۵].

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 [۱۲]. مطالعات نشان می‌دهند که درصد قابل‌توجهی از خطاهای به‌وجودآمده پس از بازمهندسی، ناشی از برداشت اشتباه از منطق کسب‌وکار و وابستگی‌های داخلی سیستم است [۱۳]. 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 [۱۱]. 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 [۸]. این وضعیت باعث می‌شود که آسیب‌پذیری‌های شناخته‌شده برای مدت طولانی در سیستم باقی بمانند و مهاجمان بتوانند از آن‌ها سوءاستفاده کنند [۷]. 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 [۳]. یکی دیگر از چالش‌های امنیتی، ضعف در مدیریت وصله‌های امنیتی است. پژوهش Dissanayake و همکاران [۶] 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 [۱۵]. 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 [۱۵]. 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 [۱۵]. 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 [۱۶]. 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 [۱۵]. 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 [۱۶]. 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] From history: development Software to agile to waterfall devops، ۲۰۲۲.

[۲] Software development life cycle (sdlc). ۲۰۲۵.

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

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

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

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

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

[۸] Integrity ۳۶۰. How legacy software and hardware is a ticking cyber security risk time-bomb، ۲۰۲۳.

[۹] Watchdog. IP Reverse Engineering law: Understanding and minimize restrictions risks، ۲۰۲۱.

[۱۰] Saurabh Jha، K. Kumar and S. Kumar. From theory to practice: Understanding devops culture and mindset. Cogent Engineering، ۱۰(۱):۲۲۵۱۷۵۸، ۲۰۲۳.

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

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

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

[۱۴] ModLogix. Legacy Risks and mitigations، ۲۰۲۲.

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

[۱۶] Pascal A. et al. reverse model-driven in studies Case Proceedings In engineering. En- Knowledge Discovery، Knowledge on Conference Joint International th۱۱ the of K)،۳(IC Management Knowledge and gineering volume ۲، pages ۷۳۱–۷۴۰، ۲۰۱۹.

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

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

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

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

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