دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی کامپیوتر

گزارش گروهی در تکامل نرم افزار

نویسندگان:

محمدسینا اله‌کرم	**محمد اکبرپورجنت**	**محمد شمس الدینی**
رضا لشنی زند	**ارسلان واثق**	**علیرضا پرستار**
مهدوی اصل	**رضا قنبرزاده**	**معراج سوسن**
بهنام کاظمی	**فاطمه عاصی آتشکاهی**	**محمد شیبانی**
نام نویسنده سیزدهم	**نام نویسنده چهاردهم**	

استاد راهنما: دکتر محمدهادی علائیان

۱۶ آبان ۱۴۰۴

چکیده

تحولات مهندسی نرم‌افزار از روش‌های ابتدایی بدون ساختار به مدل‌های خطی مانند آبشاری و سپس به رویکردهای تکرارشونده و چابک، نشان‌دهنده تلاش برای مدیریت پیچیدگی و افزایش کیفیت سیستم‌ها بوده است. با این حال، چالش‌هایی مانند ارتباطات ناکارآمد، مستندسازی ضعیف، انباشت بدهی فنی و ناسازگاری با فناوری‌های نوین، همچنان تهدیدی برای پایداری نرم‌افزارها محسوب می‌شوند.

DevOps به‌عنوان فرهنگی نوین و مجموعه‌ای از ابزارها، با هدف یکپارچه‌سازی تیم‌های توسعه و عملیات، افزایش سرعت تحویل و ارتقای کیفیت معرفی شد. خودکارسازی فرآیندهای CI/CD و استفاده از ابزارهایی مانند ،Kubernetes Docker و ،Jenkins امکان تحویل سریع، کاهش خطا و افزایش پایداری استقرار را فراهم می‌کند. با این حال، چالش‌هایی مانند پیچیدگی زیرساخت و مقاومت فرهنگی نیازمند آموزش، مستندسازی و فرهنگ‌سازی هستند.

بازطراحی (Reengineering) نرم‌افزار برای سیستم‌های قدیمی ضروری است و دلایل آن شامل ضعف معماری، فناوری‌های منسوخ، تغییر نیازمندی‌ها و انباشت بدهی فنی است. تکنیک‌های بازآرایی (Refactoring)، مهندسی معکوس (Reverse-Engineering) و مهاجرت افزایشی (Incremental Migration) ضمن افزایش قابلیت نگهداری، هزینه اصلاح بدهی فنی را کاهش می‌دهند و ریسک تغییرات را مدیریت می‌کنند. مطالعات موردی مانند بازطراحی اپلیکیشن PayPal و نئوبانک فوربیکس، نشان‌دهنده تاثیر مستقیم بازطراحی بر تجربه کاربری، یکپارچگی خدمات و مزیت رقابتی هستند.

در نهایت، ترکیب رویکردهای DevOps و بازطراحی، پایداری و تکامل نرم‌افزارها را تضمین می‌کند. توصیه می‌شود تیم‌های مهندسی بر فرهنگ همکاری، مدیریت فعال بدهی فنی، مستندسازی همگام با توسعه، خودکارسازی CI/CD و آموزش مستمر تمرکز کنند. همچنین، مسیرهای تحقیقاتی آینده شامل امنیت یکپارچه در DevOps ،(DevSecOps) کاربرد هوش مصنوعی در مهندسی معکوس، مدیریت پیچیدگی ابزارها، آموزش مهارت‌های نرم و توسعه الگوهای پیشرفته مهاجرت و بازطراحی خواهد بود.

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

۲.۲ 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 [۳].

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.

۱.۲.۲ تمایز آن با Reengineering و Refactoring

- **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.

۳.۲ مرور ادبیات

۴.۲ 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 [۳].

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.

۱.۴.۲ تمایز آن با Reengineering و Refactoring

- **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.

۵.۲ روش‌های موجود

۶.۲ مشکلات فنی

چالش‌های فنی از مهم‌ترین عوامل تأخیر در توسعه و افزایش هزینه‌های نگهداری نرم‌افزار به شمار می‌روند. این مشکلات معمولاً ریشه در ساختار پیچیده‌ی کد، فناوری‌های قدیمی، یا ضعف در طراحی اولیه دارند. هرچه پروژه بزرگ‌تر و طولانی‌تر باشد، احتمال بروز چنین مشکلاتی بیشتر می‌شود.

۱.۶.۲ بدهی فنی (Technical Debt)

بدهی فنی به تصمیمات کوتاه‌مدتی اشاره دارد که برای سرعت‌بخشیدن به تحویل محصول گرفته می‌شوند، اما در آینده هزینه‌های سنگینی به پروژه تحمیل می‌کنند. این بدهی می‌تواند شامل کدهای

تکراری، طراحی ناقص، تست‌های ناکافی، یا معماری غیراستاندارد باشد. هر بار که تیمی به‌جای اصلاح ریشه‌ای مشکلات، راه‌حل موقتی انتخاب می‌کند، بدهی فنی افزایش می‌یابد.

۲.۶.۲ ناسازگاری با فناوری‌های جدید

سیستم‌هایی که بر پایه فناوری‌های منسوخ بنا شده‌اند، به‌تدریج دچار محدودیت عملکردی و امنیتی می‌شوند. مهاجرت به فناوری‌های جدید اغلب دشوار، هزینه‌بر و زمان‌گیر است. راهکار مؤثر، استراتژی «مدرن‌سازی تدریجی» است؛ یعنی انتقال گام‌به‌گام بخش‌های حیاتی سیستم به فناوری‌های جدید بدون توقف کامل سیستم.

۳.۶.۲ خطاهای طراحی و ماژول‌های ناسازگار

طراحی ناپایدار و غیرماژولار باعث بروز خطاهای میان‌ماژولی، افت کارایی، اختلال در عملکرد و دشواری شدید در افزودن ویژگی‌های جدید می‌شود. استفاده از معماری مدرن مانند microservices و طراحی مبتنی بر API می‌تواند از بروز تضادها جلوگیری کند.

۷.۲ مشکلات انسانی

عوامل انسانی شاید کم‌اهمیت‌تر از مشکلات فنی یا سازمانی به نظر برسند، اما در واقع یکی از تأثیرگذارترین عوامل بر موفقیت پروژه‌های نرم‌افزاری محسوب می‌شوند. نرم‌افزار توسط انسان‌ها طراحی، توسعه و نگهداری می‌شود، و کیفیت عملکرد انسانی مستقیماً بر کیفیت محصول نهایی اثر می‌گذارد.

۱.۷.۲ فرسودگی تیم

تحویل‌های پی‌درپی، ساعات کاری طولانی، فشار زمان و عدم توازن میان زندگی شخصی و کاری باعث فرسودگی شغلی می‌شود. فرسودگی منجر به کاهش تمرکز، افت کیفیت کد و افزایش نرخ خروج کارکنان می‌شود. فرهنگ کاری سالم نقش اساسی در حفظ پایداری نیروی انسانی دارد.

۲.۷.۲ فقدان مهارت‌های جدید

سرعت بالای پیشرفت فناوری باعث می‌شود سازمان‌هایی که در آموزش نیروی انسانی سرمایه‌گذاری نکنند، دیر یا زود با بحران مهارت مواجه شوند. کارگاه‌های آموزشی، اشتراک دانش و استفاده از مشاورهای فنی به حفظ انگیزه و رشد مهارتی کمک می‌کنند.

مشکلات در نگهداری و تکامل سیستم‌های قدیمی (Legacy Systems)

Systems Legacy به نرم‌افزارها یا سیستم‌های قدیمی گفته می‌شود که هنوز در سازمان‌ها و شرکت‌ها مورد استفاده قرار می‌گیرند، اما با فناوری‌های مدرن سازگار نیستند یا از استانداردهای فعلی فاصله دارند؛ به عبارت ساده‌تر، Legacy Systems سیستم‌هایی هستند که عملکردشان هنوز ادامه دارد، اما به دلیل قدیمی بودن فناوری، پیچیدگی کد یا کمبود مستندات، نگهداری و تکامل آن‌ها با چالش‌های قابل توجهی همراه است. به دلیل اینکه بسیاری از زیرساخت ها و نرم افزارهای سازمان ها و شرکت ها را تشکیل می‌دهند، از اهمیت بالایی برخوردار هستند. در ادامه به چند مورد از دلایل اهمیت آن می‌پردازیم:

پایداری عملیاتی سازمان: بسیاری از فرآیندهای حیاتی سازمان‌ها به Legacy Systems وابسته هستند. توقف یا خرابی این سیستم‌ها می‌تواند باعث اختلال جدی در عملکرد سازمان شود.

حفظ سرمایه‌گذاری‌های گذشته: توسعه و پیاده‌سازی سیستم‌های بزرگ و پیچیده نیازمند هزینه و زمان زیادی بوده است. نگهداری این سیستم‌ها به سازمان‌ها اجازه می‌دهد سرمایه‌گذاری‌های قبلی خود را حفظ کنند و از دوباره‌کاری جلوگیری شود.

تداوم خدمات به کاربران: سیستم‌های قدیمی غالباً به‌صورت مستقیم با کاربران نهایی یا مشتریان سروکار دارند. نگهداری این سیستم‌ها باعث می‌شود کیفیت خدمات و رضایت کاربران حفظ شود.

تسهیل تکامل تدریجی: Systems Legacy می‌توانند پایه‌ای برای تکامل نرم‌افزار و افزودن قابلیت‌های جدید باشند. با نگهداری و بازسازی تدریجی، می‌توان آن‌ها را با فناوری‌های جدید یکپارچه کرد.

کاهش ریسک‌های عملیاتی: جایگزینی یک سیستم قدیمی با نرم‌افزار جدید همیشه پرریسک است. نگهداری و بهبود سیستم‌های موجود، ریسک‌های مربوط به مهاجرت و بازنویسی کامل را کاهش می‌دهد.

مشکلات نگهداری و تکامل سیستم های قدیمی (Legacy Systems)

- **پیچیدگی و ساختار قدیمی کد:** نرم‌افزارهای Legacy اغلب با فناوری‌ها و الگوهای قدیمی توسعه یافته‌اند که ساختار کد را پیچیده می‌کند و درک و اصلاح آن‌ها برای توسعه‌دهندگان جدید دشوار است.
- **وابستگی به فناوری‌های منسوخ:** زبان‌های برنامه‌نویسی، پایگاه داده‌ها و سیستم‌های عامل قدیمی دیگر پشتیبانی نمی‌شوند.
- **کمبود مستندات و دانش فنی:** مستندات کامل و به‌روز اغلب وجود ندارد و خروج افراد با تجربه باعث از بین رفتن دانش کلیدی می‌شود.
- **سازگاری محدود با سیستم‌های جدید:** سیستم‌ها معمولاً طراحی نشده‌اند تا با فناوری‌های جدید کار کنند.
- **هزینه و زمان بالا برای نگهداری و تکامل:** هر تغییر یا بهبود در سیستم‌های Legacy نیازمند تست گسترده و زمان طولانی است.

نمونه: سیستم‌های Banking Core در بانک‌های بزرگ ایران مثل بانک ملی که در دهه ۷۰ و ۸۰ با COBOL و Delphi توسعه پیدا کرده‌اند و هنوز در عملیات پایه بانکی پایدار هستند.

هزینه‌های نگهداری و تکامل

بخش نگهداری و تکامل حدود ۶۰ تا ۸۰ درصد از کل چرخه نرم‌افزار را به خود اختصاص می‌دهد:

- **هزینه‌های نیروی انسانی:** نیازمند متخصصان با تجربه در فناوری‌های منسوخ.
- **هزینه‌های تست و تضمین کیفیت:** طراحی و اجرای تست‌های عملکردی، امنیتی و یکپارچگی سیستم.
- **هزینه‌های ابزار و فناوری:** استفاده از ابزارهای مدیریت نگهداری، پایگاه داده و نسخه‌بندی.
- **هزینه‌های بدهی فنی (Technical Debt):** ناشی از تصمیمات کوتاه‌مدت توسعه.
- **هزینه‌های یکپارچه‌سازی و مهاجرت:** افزودن قابلیت‌های جدید یا مهاجرت سیستم‌های قدیمی به مدرن.

روش‌های کاهش مشکلات

مدیریت تغییرات

مدیریت تغییرات (Change Management) به مجموعه فرآیندهایی گفته می‌شود که هدف آن کنترل، مستندسازی و پیگیری تغییرات نرم‌افزار است.

اهمیت مدیریت تغییرات:

- جلوگیری از خطاهای ناشی از تغییرات غیرمستند یا غیرکنترل‌شده.
- تضمین سازگاری تغییرات با سیستم‌های موجود و فرآیندهای سازمان.
- امکان پیگیری و بازگشت به نسخه‌های قبلی در صورت بروز مشکل.
- کاهش زمان و هزینه نگهداری با شناسایی سریع مشکلات.

اصول مدیریت تغییرات:

- ثبت و مستندسازی تغییرات.
- بررسی و تأیید تغییرات قبل از اعمال.
- تست پیش از اجرا (واحد، یکپارچگی، عملکرد).
- پیگیری و گزارش‌دهی پس از اعمال تغییرات.
- بازگشت به نسخه قبلی (Rollback Plan).

Refactoring و بازطراحی جزئی

Refactoring به فرآیند بازسازی کد نرم‌افزار بدون تغییر رفتار خارجی آن گفته می‌شود.

اهمیت Refactoring:

- کاهش پیچیدگی و افزایش خوانایی کد.
- کاهش خطاهای نرم‌افزاری.

- افزایش انعطاف‌پذیری سیستم.
- کاهش هزینه نگهداری در بلندمدت.

اصول :Refactoring

- تغییر تدریجی و بخش‌بخش.
- تست مستمر قبل و بعد از هر تغییر.
- مستندسازی تغییرات.
- استفاده از الگوهای طراحی و استانداردهای کدنویسی.

Integration Continuous (ادغام مداوم)

ادغام مداوم یک رویکرد در مهندسی نرم‌افزار است که توسعه‌دهندگان به‌طور مکرر تغییرات خود را در مخزن اصلی کد منبع ادغام می‌کنند. **اهمیت:**

- کشف سریع خطاها.
- کاهش هزینه‌های نگهداری.
- افزایش کیفیت نرم‌افزار.
- سهولت در تکامل نرم‌افزار.

اصول و ابزارهای ادغام مداوم:

- مخزن مشترک کد منبع .(GitHub/GitLab)
- تست خودکار پس از هر .Commit
- استفاده از سرور CI، Jenkins) GitLab CI/CD، Travis CI، GitHub Actions).
- بازخورد سریع به توسعه‌دهندگان.

مطالعه‌ی موردی از شکست پروژه‌ها

سیستم LASCAD (London Aided Computer Service Ambulance Dispatch)

LASCAD پروژه‌ای بود که در اوایل دهه ۱۹۹۰ توسط خدمات آمبولانس لندن اجرا شد. هدف خودکارسازی فرآیند دریافت تماس‌های اضطراری، تخصیص آمبولانس و پیگیری عملیات امداد بود؛ اما سیستم عملاً از کار افتاد.

دلایل شکست:

- طراحی ضعیف و غیرقابل نگهداری.
- تست ناکافی.
- نبود مدیریت تغییرات.
- مستندسازی و آموزش ضعیف.
- فشار زمانی و مدیریتی.

نتایج و پیامدها: سیستم چند ساعت پس از راه‌اندازی به‌طور کامل از کار افتاد و اعتبار سازمان کاهش یافت.

پروژه‌ی Virtual File Case در FBI

در سال ۲۰۰۰، سازمان FBI تصمیم گرفت سیستم‌های قدیمی خود را با یک سیستم مدرن جایگزین کند.

دلایل شکست:

- زیرساخت قدیمی و ناسازگار.
- مدیریت تغییرات ضعیف.
- طراحی غیرقابل نگهداری.
- فقدان ارتباط مؤثر میان ذینفعان.

- فشار زمانی و مدیریتی.

نتایج و پیامدها: پروژه پس از صرف حدود ۱۷۰ میلیون دلار و چهار سال، کنار گذاشته شد و FBI پروژه جدیدی آغاز کرد.

درس‌ها: سیستم‌های قدیمی بدون مستندات مناسب، ریسک بالایی دارند. مدیریت تغییرات و نیازمندی‌ها باید از روز اول برقرار باشد.

جمع‌بندی فصل

چرخه‌ی توسعه و تکامل نرم‌افزار، فرآیندی پویا و مداوم است. تجربه نشان داده بخش عمده‌ای از چالش‌ها و شکست‌ها در مراحل نگهداری و تکامل رخ می‌دهد.

مشکلات رایج:

- کدهای پیچیده و غیرمستند.
- نبود مدیریت تغییرات مؤثر.
- افزایش هزینه‌های نگهداری.
- مشکلات فنی ناشی از Systems. Legacy
- کمبود تست‌های مداوم و خودکار.
- ضعف ارتباط میان تیم‌ها و کاربران نهایی.

نگهداری و تکامل نرم‌افزار بخش اصلی چرخه‌ی حیات نرم‌افزار است و سازمان‌هایی که به طراحی منعطف، مستندسازی دقیق، مدیریت تغییرات، تست مستمر و نوسازی سیستم‌های قدیمی توجه کنند، قادر به کاهش هزینه و افزایش طول عمر سیستم‌ها خواهند بود.

فصل ۳

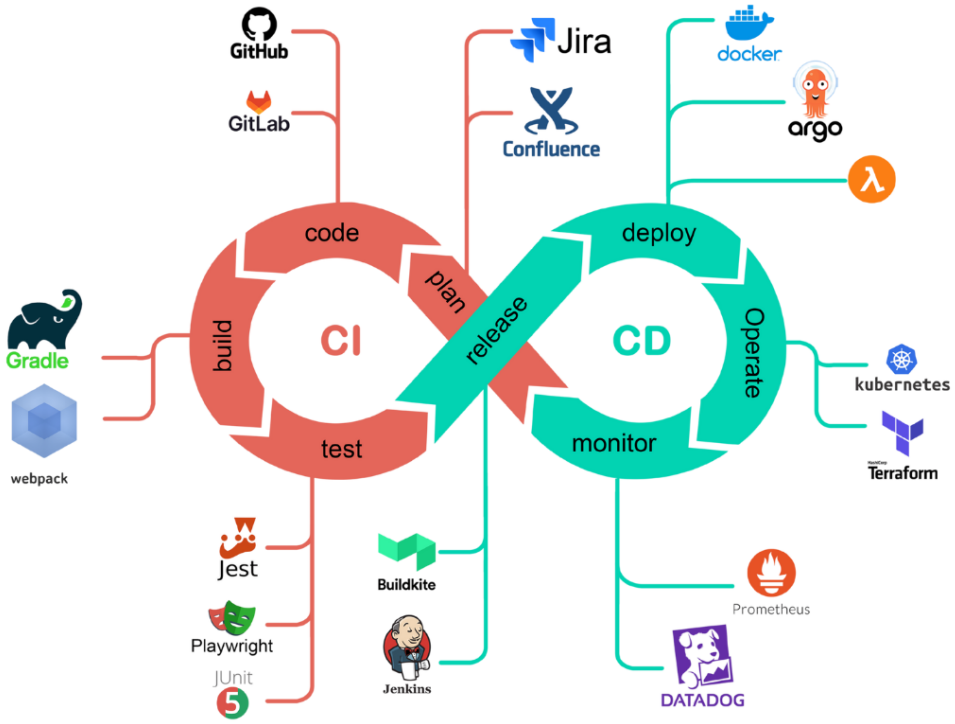
DevOps و نقش آن در فرایند تکامل نرم‌افزار

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 (Continues 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



شکل ۱.۳: tools and life-cycle DevOps

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، Stack Elastic 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 [۴]، 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 [۴] آمده است، هم‌ترازی هدف‌ها باعث می‌شود معیارهای ارزیابی از فردمحور به تیم‌محور تغییر کند.

شکل ۲.۳: نمایی از مؤلفه‌های فرهنگ و ذهنیت DevOps بر اساس [۴].

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 [۴] 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 [۴] 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 [۴] 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 [۴]، 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 [۴] 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 [۴] 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 [۴] 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 [۴]، 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 [۴] 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 که هدف نهایی آن‌ها افزایش کارایی، قابلیت نگهداری و طول عمر نرم‌افزار است. سه مورد کلیدی در این حوزه عبارتند از:

۱.۵.۴ بازآرایی (Refactoring)

بازآرایی فرآیند بازسازی (restructuring) ساختار داخلی کد بدون تغییر رفتار خارجی آن است. هدف اصلی بهبود طراحی، خوانایی و قابلیت نگهداری کد است که در نهایت توسعه‌ی ویژگی‌های جدید را آسان‌تر می‌کند. این کار اغلب با انجام تغییرات کوچک و مطمئن مانند تغییر نام متغیرها، استخراج متدها و ساده‌سازی شرط‌ها انجام می‌شود. ابزارهای مدرن یکپارچه در محیط‌های توسعه (IDE) مانند قابلیت‌های بازآرایی در IDEA IntelliJ یا ReSharper برای ،C# این فرآیند را به صورت خودکار و ایمن انجام می‌دهند [**ibm-refactoring**].

۲.۵.۴ مهندسی معکوس (Reverse Engineering)

مهندسی معکوس فرآیند استخراج طراحی، معماری و مشخصات یک سیستم از کد منبع موجود است. این تکنیک به ویژه برای درک سیستم‌های legacy که فاقد مستندات کافی هستند، حیاتی است. ابزارهای این حوزه مانند ابزارهای تولید نمودارهای UML از کد (مانند Architect Enterprise یا ابزارهای موجود در Visual Studio) یا دیس اسمبلرها، لایه‌های مختلف سیستم را آشکار کرده و درک آن را برای تیم‌های توسعه ممکن می‌سازند [**geeks-reverse-engineering**].

۳.۵.۴ مهاجرت (Migration)

مهاجرت به فرآیند انتقال یک سیستم نرم‌افزاری از یک محیط تکنولوژیکی قدیمی به یک محیط جدیدتر و قدرتمندتر اطلاق می‌شود. این امر می‌تواند شامل مهاجرت پایگاه داده (مانند انتقال از Oracle به ،PostgreSQL) مهاجرت سکو (مانند انتقال یک برنامه از ویندوز به وب) یا حتی مهاجرت زبان برنامه‌نویسی باشد. ابزارهای اتوماسیون این فرآیند، ریسک و تلاش انسانی مورد نیاز را به شدت کاهش می‌دهند. موفقیت این فرآیند وابسته به برنامه‌ریزی دقیق، اجرای مرحله‌ای و تست گسترده برای اطمینان از حفظ یکپارچگی داده‌ها و عملکرد سیستم است [**geeks-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 (Reliability)، افزایش امنیت، بهبود قابلیت نگهداری (Maintainability) و افزایش قابلیت گسترش (Scalability) 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 اپلیکیشن و هویت بصری متمرکز است، در حالی که سیستم بانکی به معرفی یک نئوبانک کسب‌وکاری داخلی می‌پردازد. در ادامه، این دو مورد به صورت جداگانه ارائه شده‌اند:

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" شخصی‌شده برای کاربران بود. مشکلات اصلی که این بازطراحی به دنبال رفع آن‌ها بود، شامل صفحه اصلی غیرجذاب، سردرگمی کاربران در پیمایش و عدم اطلاع‌رسانی شفاف بود. راه‌حل‌های کلیدی اجرا شده در این بازطراحی عبارتند از:

- **تجربه شخصی‌سازی‌شده**: صفحه اصلی به فضایی شخصی برای مدیریت امور مالی کاربر تبدیل شد. با نمایش فعالیت‌های حساب و بینش‌های مالی مفید، حس تعلق کاربر به اپلیکیشن تقویت شد.
- **پیمایش ساده‌شده**: نوار پیمایش پایین اپلیکیشن با آیکون‌ها و برچسب‌های قابل درک طراحی شد تا کاربران به راحتی به ویژگی‌های مورد نظر خود دسترسی پیدا کنند.
- **شفافیت اطلاعاتی**: اطلاعات حیاتی مانند موجودی حساب و کارت در حال استفاده، به وضوح و در صفحه اصلی نمایش داده می‌شوند تا از سردرگمی کاربر کاسته شود.
- **تجمیع عملکردها**: عملکردهای مرتبط با پول و حساب در یک بخش گروه‌بندی شدند تا قابلیت کشف و استفاده از آن‌ها برای کاربر آسان‌تر شود.

این تغییرات منجر به ایجاد تجربه‌ای شد که با انتظارات کاربران مطابقت بیشتری دارد، قابلیت کشف ویژگی‌ها را بهبود بخشید و از طریق شفافیت، اعتماد کاربران را افزایش داد. علاوه بر این، پی‌پال هویت بصری برند خود را نیز به‌روز کرد که شامل طراحی قلم سفارشی "PayPal Pro" و ساده‌سازی پالت رنگی برای نمایشی مدرن‌تر و خوش‌بینانه‌تر بود.

۲.۷.۴ بازطراحی بانکداری برای کسب‌وکارهای کوچک (فوربیکس)

در ایران، نمونه بارز بازطراحی در سیستم بانکی، ظهور "نئوبانک‌های کسب‌وکاری" مانند فوربیکس است. هدف فوربیکس، بازطراحی خدمات بانکی برای پاسخگویی به نیازهای خاص کسب‌وکارهای کوچک و متوسط بود که اغلب توسط سیستم بانکی سنتی نادیده گرفته می‌شوند. ویژگی‌های کلیدی این بازطراحی شامل:

- **یکپارچگی خدمات**: فوربیکس خدمات بانکی (مانند افتتاح حساب و درگاه پرداخت) را با ابزارهای عملیاتی کسب‌وکار (مانند سیستم حسابداری، صدور فاکتور، CRM و مدیریت انبار) در یک پلتفرم واحد ادغام کرد.
- **تمرکز بر کاربرپسندی**: این پلتفرم با ارائه یک اپلیکیشن موبایل با رابط کاربری ساده، تجربه مالی ساده‌ای را برای صاحبان کسب‌وکارها فراهم می‌کند.
- **اتوماسیون برای صرفه‌جویی در زمان**: با اتصال خودکار تراکنش‌های بانکی به سیستم حسابداری، فرآیندهای دستی کاهش یافته و تا ۴۰ درصد در زمان صرفه‌جویی می‌شود.

این رویکرد یکپارچه، چالش‌های کسب‌وکارها در استفاده همزمان از سیستم‌های ناهمگون بانکی و حسابداری را برطرف کرده و مدیریت امور مالی و عملیاتی را برای آن‌ها بسیار کارآمدتر کرده است.

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 [۳].

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 [۳].

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 [۳]. 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].

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 [۵]. 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 [۶]. 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 [۷]. 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 [۸]. 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 [۹]. 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 [۱۰]. 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 [۱۲].

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 [۱۳]. مطالعات نشان می‌دهند که درصد قابل‌توجهی از خطاهای به‌وجودآمده پس از بازمهندسی، ناشی از برداشت اشتباه از منطق کسب‌وکار و وابستگی‌های داخلی سیستم است [۱۴]. 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 [۱۵]. 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 [۱۶]. این وضعیت باعث می‌شود که آسیب‌پذیری‌های شناخته‌شده برای مدت طولانی در سیستم باقی بمانند و مهاجمان بتوانند از آن‌ها سوءاستفاده کنند [۱۷]. 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 [۱۸]. یکی دیگر از چالش‌های امنیتی، ضعف در مدیریت وصله‌های امنیتی است. پژوهش Dissanayake و همکاران [۱۹] 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 [۲۰]. 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 [۲۰]. 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 [۲۰]. 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 [۲۱]. 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 Zhao و Moraga، 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 [۲۰]. 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 ه‌م‌کاران و، 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 [۲۱]. 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.

فصل ۶

فایل های PE

۱.۶ تفاوت آدرس و آفست

۱.۱.۶ تعریف آدرس (Address)

آدرس به مکان منحصر به فردی در حافظه اصلی (RAM) اشاره دارد که یک واحد داده (معمولاً یک بایت) در آن ذخیره شده است. پردازنده (CPU) برای دسترسی به داده‌ها یا دستورالعمل‌ها، از این آدرس‌ها استفاده می‌کند.

- در معماری‌های قدیمی‌تر مانند Intel ۸۰۸۶، آدرس‌ها به صورت **آدرس فیزیکی** (Physical Address) و در معماری‌های مدرن‌تر، اغلب به صورت **آدرس مجازی** (Virtual Address) به برنامه ارائه می‌شوند که توسط واحد مدیریت حافظه (MMU) به آدرس فیزیکی ترجمه می‌شوند.
- آدرس‌دهی مستقیم و مطلق، مستقیماً به یک مکان خاص در حافظه اشاره می‌کند.

۲.۱.۶ تعریف آفست (Offset)

آفست (جابجایی یا فاصله) بیانگر **فاصله** یک بایت داده یا یک دستورالعمل، از یک نقطه شروع مشخص (معمولاً ابتدای یک ساختار، آرایه، رکورد، یا سگمنت در معماری‌های سگمنتی) است.

- در معماری‌های سگمنتی (مانند Mode Real پردازنده ۸۰۸۶)، آدرس کامل یک مکان حافظه (**آدرس فیزیکی**) از ترکیب یک آدرس پایه (Base Address) که در ثبات سگمنت ذخیره شده، و

آفست به دست می‌آید.

- آدرس فیزیکی P از طریق فرمول زیر محاسبه می‌شود (در معماری ۸۰۸۶):

$$P = (\text{Base Segment} \times ۱۶) + \text{Offset}$$

- آفست، در واقع، یک آدرس **نسبی** Relative) (Address است.

۳.۱.۶ کاربرد در تحلیل باینری و مهندسی معکوس

در تحلیل باینری Binary) (Analysis و مهندسی معکوس Reverse) ،(Engineering درک آدرس و آفست از اهمیت حیاتی برخوردار است.

آدرس‌ها

- **نقشه‌برداری حافظه:** آدرس‌های مجازی و فیزیکی برای تعیین محل قرارگیری توابع، داده‌ها و متغیرها در هنگام اجرای برنامه ضروری هستند.
- **Pointer): (Instruction EIP/RIP** ثبات اشاره‌گر دستورالعمل EIP) در ۳۲ بیت و RIP در ۶۴ بیت) همواره آدرس دستوری را که CPU قرار است اجرا کند، نگهداری می‌کند. تحلیلگر با بررسی تغییرات این ثبات می‌تواند مسیر اجرای برنامه Control) (Flow را دنبال کند.
- **Breakpoints:** برای توقف اجرای برنامه در یک نقطه خاص (مثلاً ابتدای یک تابع مشکوک)، نیاز است که آدرس دقیق آن مکان در حافظه مشخص شود.

آفست‌ها

- **آدرس مجازی نسبی :(RVA)** در ساختار فایل‌های اجرایی مانند PE (Windows) و ELF ،(Linux) بسیاری از آدرس‌ها به صورت RVA Virtual) (Relative Address) ذخیره می‌شوند که در واقع آفست نسبت به آدرس مبدأ بارگذاری فایل در حافظه Image) (Base هستند. این امر جابه‌جایی فایل اجرایی را در حافظه (ASLR) آسان می‌کند.
- **تجزیه ساختارها:** آفست‌ها برای دسترسی به فیلدهای یک ساختار داده (مانند ساختارهای ویندوز، یا شیءها در برنامه‌نویسی شیءگرا) ضروری هستند. برای مثال، برای دسترسی به فیلد دوم یک ساختار، باید آفست آن فیلد نسبت به ابتدای ساختار مشخص شود.

- **Overflow: Buffer** در تحلیل آسیب‌پذیری‌های سرریز بافر (Buffer Overflow)، تعیین آفست دقیق برای رونویسی آدرس بازگشت (Return Address) یا سایر داده‌های حساس (مانند اشاره‌گرهای پشته) یک مرحله کلیدی است.

۴.۱.۶ مثال‌های عددی

مثال ۱: آدرس‌دهی سگمنتی (۸۰۸۶)

فرض کنید ثبات سگمنت داده (DS) حاوی مقدار $۱۰۰۰h$ و ثبات اندیس مبدأ (SI) (که به عنوان آفست عمل می‌کند) حاوی مقدار $۰۰A۰h$ باشد.

- **Base Segment (شیفت‌یافته):** $۱۰۰۰h \times ۱۰h = ۱۰۰۰۰h$
- **Offset:** $۰۰A۰h$
- **آدرس فیزیکی:**

$$P = ۱۰۰۰۰h + ۰۰A۰h = ۱۰۰A۰h$$

دستور اسمبلی مانند `MOV AL, [SI]` داده‌ای که در آدرس فیزیکی $۱۰۰A۰h$ قرار دارد را به ثبات AL منتقل می‌کند.

مثال ۲: محاسبه آفست در آرایه

در زبان اسمبلی (یا در زبان‌های سطح بالا مانند C فرض کنید یک آرایه از اعداد صحیح ۴ بایتی `int` به نام `array my` در آدرس پایه $B = ۰x۴۰۰۰۰۰$ تعریف شده باشد. برای دسترسی به عنصر شماره i (که اندیس آن از صفر شروع می‌شود)، از آفست استفاده می‌شود.

- **فرمول آفست:** $\text{Offset} = i \times \text{Element of Size}$
- **آدرس عنصر سوم ($i = ۲$):**

$$\text{Address}(۲) = B + (۲ \times ۴) = ۰x۴۰۰۰۰۰ + ۸ = ۰x۴۰۰۰۰۸$$

- در اینجا، مقدار ۸ (بایت) آفست عنصر سوم نسبت به ابتدای آرایه است.

۵.۱.۶ نکات کاربردی در مهندسی معکوس

۱. **محاسبه RVA:** در مهندسی معکوس، اغلب نیاز است که آدرس‌های مجازی (VA) را به آفست‌های فایل (File Offsets) تبدیل کنید تا بتوانید داده‌ها را در فایل باینری روی دیسک مشاهده یا تغییر دهید. این کار با استفاده از جدول سِکشن‌ها (Section Table) در هدر فایل PE/ELF انجام می‌شود.

۲. **آدرس‌دهی نسبی به RIP:** در معماری‌های ۶۴ بیتی (x۶۴)، آدرس‌دهی نسبی به ثبات RIP رایج است: `MOV EAX, [RIP + offset]`. تحلیلگر باید مقدار آفست را به آدرس فعلی RIP اضافه کند تا آدرس مقصد را پیدا کند. این امر به ویژه در کدهای مستقل از موقعیت (PIC) بسیار مهم است.

۳. **آفست‌های پشته:** در هنگام بررسی توابع، متغیرهای محلی و پارامترهای تابع با استفاده از آفست‌هایی نسبت به ثبات پایه پشته (RBP/EBP) یا اشاره‌گر پشته (RSP/ESP) آدرس‌دهی می‌شوند. پیدا کردن آفست متغیرها نسبت به RBP (مانند `[RBP - 0x20]`) برای درک منطق تابع حیاتی است.

فصل ۷

دیباگ (Debugging) و اشکال‌زداها (Debuggers)

۱.۷ انواع دیباگرها: دسته‌بندی از نظر سطح کارکرد

دیباگرها از حیث سطح دسترسی به سیستم عامل به دو دسته اصلی تقسیم می‌شوند که هر یک کاربردها و قابلیت‌های متمایزی دارند.

۱.۱.۷ دیباگرهای حالت کاربر (User-Mode Debuggers)

این دسته از دیباگرها در فضای کاربر اجرا شده و تنها به منابع و processهای متعلق به کاربر دسترسی دارند.

مثال‌های کاربردی

- GDB (GNU Debugger): دیباگر استاندارد برای برنامه‌های لینوکس
- WinDbg (User Mode): برای دیباگ برنامه‌های ویندوزی
- LLDB: دیباگر مدرن برای مجموعه کامپایلر LLVM
- Visual Studio Debugger: محیط یکپارچه برای برنامه‌های .NET.

مزایا

- **امنیت بالا**: به دلیل محدودیت دسترسی به هسته سیستم عامل
- **پایداری**: خطا در دیباگر باعث crash سیستم نمی‌شود
- **سهولت استفاده**: معمولاً رابط کاربری گرافیکی دارند
- **قابلیت حمل**: روی سیستم‌های مختلف قابل اجرا هستند

معایب

- **محدودیت دسترسی**: نمی‌توانند درایورها یا هسته را دیباگ کنند
- **محدودیت در بررسی حافظه**: فقط به فضای حافظه process کاربر دسترسی دارند
- **عدم توانایی در تحلیل** interrupt**ها**: قادر به دیباگ وقفه‌های سیستمی نیستند

۲.۱.۷ **دیباگرهای حالت هسته** (Kernel-Mode Debuggers)

این دیباگرها با سطح دسترسی بالاتری عمل کرده و مستقیماً با هسته سیستم عامل در ارتباط هستند.

مثال‌های کاربردی

- WinDbg (Kernel Mode): برای دیباگ درایورهای ویندوز
- KGDB: دیباگر هسته لینوکس
- SoftICE: دیباگر معروف برای سیستم‌های قدیمی

مزایا

- **دسترسی کامل**: توانایی دیباگ کل سیستم شامل هسته و درایورها
- **قدرت تشخیص بالا**: می‌توانند مسائل پیچیده سیستمی را تحلیل کنند
- **امکان دیباگ** low-level: قادر به دیباگ در سطح سخت‌افزار هستند

- **تحلیل** crash dump: توانایی تحلیل کامل dumpهای سیستمی

معایب

- **پیچیدگی**: نیاز به دانش عمیق از سیستم عامل و سخت‌افزار
- **ریسک بالا**: خطا در دیباگر می‌تواند باعث crash سیستم شود
- **مشکلات امنیتی**: دسترسی بالا می‌تواند تهدید امنیتی ایجاد کند
- **نیاز به تنظیمات خاص**: معمولاً نیاز به پیکربندی پیچیده دارند

۲.۷ دسته‌بندی دیباگرها از نظر رویکرد

دیباگرها از نظر رویکرد و روش دیباگ نیز به دسته‌های مختلفی تقسیم می‌شوند که هر کدام برای سناریوهای خاصی مناسب هستند.

۱.۲.۷ دیباگرهای نمادین (Symbolic Debuggers)

این دیباگرها از اطلاعات نمادین (symbol) برای نمایش متغیرها و توابع استفاده می‌کنند.

مثال‌های کاربردی

- GDB با اطلاعات دیباگ: با فایل‌های DWARF/PDB
- Visual Studio Debugger: با اطلاعات PDB
- LLDB با اطلاعات نمادین: برای برنامه‌های C/C++

مزایا

- **قابلیت خوانایی بالا**: نمایش نام متغیرها و توابع به جای آدرس‌های حافظه
- **عیب‌یابی سریعتر**: امکان تنظیم breakpoint بر اساس نام توابع

- **تحلیل** call stack **معنادار**: نمایش نام توابع در call stack
- **پشتیبانی از** source-level debugging: نمایش کد منبع اصلی

معایب

- **نیاز به اطلاعات دیباگ**: وابستگی به فایل‌های سمبول
- **افزایش حجم برنامه**: اطلاعات دیباگ فضای اضافی مصرف می‌کنند
- **مشکلات امنیتی**: اطلاعات دیباگ می‌تواند برای مهاجمان مفید باشد

۲.۲.۷ **دیباگرهای ریموت** (Remote Debuggers)

این دیباگرها امکان دیباگ برنامه را روی سیستم دیگری فراهم می‌کنند.

مثال‌های کاربردی

- GDBServer: برای دیباگ ریموت برنامه‌های لینوکس
- Visual Studio Remote Debugger: برای دیباگ برنامه‌های ویندوزی
- Android Studio Debugger: برای دیباگ اپلیکیشن‌های اندروید

مزایا

- **دیباگ در محیط واقعی**: امکان دیباگ روی دستگاه هدف
- **عدم تأثیر بر عملکرد**: دیباگر روی سیستم جداگانه اجرا می‌شود
- **امنیت** : دیباگ سیستم‌های production بدون نصب ابزار روی آنها
- **پشتیبانی از** embedded systems: برای دیباگ سیستم‌های توکار

معایب

- **پیچیدگی تنظیمات**: نیاز به پیکربندی شبکه و ارتباط
- **مشکلات تأخیر**: تأخیر شبکه می‌تواند بر تجربه دیباگ تأثیر بگذارد
- **مشکلات اتصال**: قطعی شبکه می‌تواند فرآیند دیباگ را مختل کند

۳.۲.۷ دیباگرهای سخت‌افزاری و Assisted-Hardware

این دیباگرها از قابلیت‌های سخت‌افزاری خاص برای دیباگ استفاده می‌کنند.

مثال‌های کاربردی

- JTAG Debuggers: برای دیباگ پردازنده‌های embedded
- In-Circuit Emulators (ICE): شبیه‌سازهای سخت‌افزاری
- ARM DSTREAM: دیباگر سخت‌افزاری برای پردازنده‌های ARM

مزایا

- **دسترسی سطح پایین**: امکان دیباگ در سطح رجیسترهای پردازنده
- **دیباگ** real-time: بدون تأثیر بر timing برنامه
- **قابلیت** trace **کردن**: ثبت اجرای دستورات به صورت real-time
- **امکان دیباگ** boot code: دیباگ کدهای قبل از راه‌اندازی سیستم

معایب

- **هزینه بالا**: تجهیزات سخت‌افزاری گران‌قیمت
- **پیچیدگی فنی**: نیاز به تخصص سخت‌افزاری
- **محدودیت حمل**: تجهیزات معمولاً قابل حمل نیستند

۴.۲.۷ دیباگرهای سخت‌افزاری در مقابل دیباگرهای نرم‌افزاری

این دسته‌بندی به ابزارهای مورد استفاده برای دیباگ اشاره دارد که هر کدام مزایا و محدودیت‌های خاص خود را دارند.

۵.۲.۷ دیباگرهای نرم‌افزاری (Software Debuggers)

این دیباگرها کاملاً مبتنی بر نرم‌افزار بوده و از قابلیت‌های سیستم عامل و خود برنامه برای دیباگ استفاده می‌کنند.

مثال‌های کاربردی

- Visual Studio Debugger: برای برنامه‌های .NET. و C++
- GDB/LLDB: برای برنامه‌های لینوکس و macOS
- Eclipse Debugger: برای برنامه‌های جاوا
- Chrome DevTools: برای دیباگ برنامه‌های وب

مزایا

- **هزینه پایین**: معمولاً رایگان یا کم‌هزینه هستند
- **دسترسی آسان**: به راحتی قابل دانلود و نصب هستند
- **یادگیری آسان**: رابط کاربری معمولاً ساده‌تر است
- **انعطاف‌پذیری**: قابلیت سفارشی‌سازی و توسعه دارند

معایب

- **محدودیت در دسترسی**: نمی‌توانند به سطوح پایین سخت‌افزار دسترسی پیدا کنند
- **تأثیر بر عملکرد**: ممکن است بر عملکرد برنامه تأثیر بگذارند
- **محدودیت در** real-time debugging: برای سیستم‌های بلادرنگ مناسب نیستند

۶.۲.۷ دیباگرهای سخت‌افزاری (Hardware Debuggers)

این دیباگرها از تجهیزات سخت‌افزاری خاص برای نظارت و کنترل اجرای برنامه استفاده می‌کنند.

مثال‌های کاربردی

- JTAG Probes: برای دیباگ میکروکنترلرها
- Logic Analyzers: برای تحلیل سیگنال‌های دیجیتال
- In-Circuit Emulators: برای شبیه‌سازی سخت‌افزار
- ARM DSTREAM/ULINK: برای پردازنده‌های ARM

مزایا

- **دسترسی کامل**: امکان مشاهده وضعیت واقعی سخت‌افزار
- **دیباگ غیرتهاجمی**: بدون تأثیر بر timing برنامه
- **قابلیت** trace **پیشرفته**: ثبت اجرای دستورات در حافظه داخلی
- **امکان دیباگ** boot code: دیباگ از اولین دستورات پردازنده

معایب

- **هزینه بسیار بالا**: تجهیزات معمولاً گران‌قیمت هستند
- **پیچیدگی فنی**: نیاز به تخصص سخت‌افزاری پیشرفته
- **محدودیت** portability: تجهیزات معمولاً سنگین و غیرقابل حمل هستند

۳.۷ جداول مقایسه‌ای انواع دیباگرها

در این بخش به مقایسه سیستماتیک انواع دیباگرها در قالب جداول مقایسه‌ای می‌پردازیم.

۱.۳.۷ مقایسه دیباگرهای حالت کاربر و حالت هسته

جدول ۱.۷: مقایسه دیباگرهای حالت کاربر و حالت هسته

معیار	دیباگر حالت کاربر	دیباگر حالت هسته
سطح دسترسی	فضای کاربر	هسته سیستم عامل
امنیت	بالا - دسترسی محدود به منابع سیستم	پایین - دسترسی کامل به سیستم
پایداری	بالا - خطا باعث کرش سیستم نمی‌شود	پایین - خطا باعث کرش کامل سیستم می‌شود
میزان پیچیدگی	کم - مناسب برای برنامه‌نویسان	زیاد - نیاز به دانش سیستم عامل
کاربرد اصلی	برنامه‌های کاربردی عادی	درایورها و سرویس‌های سیستمی
مثال‌ها	GDB, Visual Studio Debugger	WinDbg Kernel, KGDB
نیاز به تخصص	برنامه‌نویسی سطح کاربر	برنامه‌نویسی و آشنایی با هسته

۲.۳.۷ مقایسه دیباگرهای نمادین و غیرنمادین

جدول ۲.۷: مقایسه دیباگرهای نمادین و غیرنمادین

معیار	دیباگر نمادین	دیباگر غیرنمادین
نمایش اطلاعات	نام متغیرها و توابع	آدرس‌های حافظه
قابلیت خوانایی	بالا - درک آسان اطلاعات	پایین - نیاز به تفسیر آدرس‌ها
نیاز به فایل دیباگ	دارد - نیاز به فایل PDB/DWARF	ندارد - مستقل از اطلاعات دیباگ
حجم برنامه	بیشتر - به دلیل اطلاعات دیباگ	کمتر - بدون اطلاعات اضافی
امنیت اطلاعات	پایین - امکان نشت اطلاعات برنامه	بالا - محافظت از ساختار برنامه
سرعت دیباگ	سریع‌تر - دسترسی مستقیم به سمبول‌ها	کندتر - نیاز به تحلیل حافظه
مثال‌ها	GDB با سمبول، VS Debugger	GDB بدون سمبول، WinDbg بدون PDB
کاربرد	توسعه و تست	آنالیز برنامه‌های Production

۳.۳.۷ مقایسه دیباگرهای ریموت و لوکال

جدول ۳.۷: مقایسه دیباگرهای ریموت و لوکال

معیار	دیباگر ریموت	دیباگر لوکال
محل اجرای دیباگر	سیستم جداگانه	همان سیستم برنامه
تأثیر بر عملکرد	کم - منابع سیستم هدف کم مصرف می‌شود	زیاد - مصرف منابع سیستم اصلی
امنیت	بالا - مناسب برای سیستم‌های Production	پایین - خطر برای سیستم اصلی
پیچیدگی تنظیمات	زیاد - نیاز به پیکربندی شبکه	کم - راه‌اندازی سریع
مشکلات شبکه	دارد - وابسته به اتصال شبکه	ندارد - مستقل از شبکه
کاربرد اصلی	سیستم‌های Production، Embedded	توسعه محلی و تست‌های اولیه
مثال‌ها	GDBServer, VS Remote Debugger	GDB محلی، VS Debugger محلی

۴.۳.۷ مقایسه دیباگرهای سخت‌افزاری و نرم‌افزاری

۴.۷ مشکلات و محدودیت‌های ابزارهای دیباگ

محدودیت‌های فنی:

۱. **عملکرد پایین در فایل‌های بزرگ**

بارگذاری نقشه‌های حافظه، تحلیل گراف فراخوانی/کنترل و به‌روزرسانی نماهای گرافیکی در پروژه‌های حجیم، به‌ویژه در IDA و x64dbg، موجب تأخیر محسوس، مصرف زیاد RAM و UI غیرپاسخ‌گو می‌شود. این مسئله در باینری‌های فشرده یا دارای تعداد زیاد DLL/so تشدید می‌شود. [۲۲، ۲۳]

۲. **نیاز به سمبل‌ها**

در غیاب PDB/DWARF یا هنگام mismatch نسخهٔ سمبل‌ها (رایج در WinDbg) شناسایی

جدول ۴.۷: مقایسه دیباگرهای سخت‌افزاری و نرم‌افزاری

معیار	دیباگر سخت‌افزاری	دیباگر نرم‌افزاری
هزینه	بسیار بالا - تجهیزات گران‌قیمت	کم تا متوسط - معمولاً رایگان
زمان راه‌اندازی	کند و پیچیده - نصب و تنظیم طولانی	سریع - نصب و اجرای آسان
میزان دسترسی	دسترسی کامل به سخت‌افزار	محدود به فضای کاربر/هسته
تأثیر بر عملکرد	غیرتهاجمی - بدون تأثیر بر timing	ممکن است تأثیر بگذارد
قابلیت Trace	پیشرفته - ثبت دقیق اجرای دستورات	محدود - وابسته به قابلیت‌های نرم‌افزار
کاربرد اصلی	سیستم‌های توکار و درایورها	برنامه‌های کاربردی
نیاز به تخصص	سخت‌افزار و الکترونیک پیشرفته	برنامه‌نویسی
مثال‌ها	JTAG, ICE, Logic Analyzer	GDB, Visual Studio, LLDB
قابلیت حمل	معمولاً کم - تجهیزات سنگین	بالا - نرم‌افزار قابل حمل

توابع، ساختارها و بازسازی Stack-Trace ناپایدار و خطاپذیر می‌شود. بهینه‌سازی‌های شدید کامپایلر (Inlining/حذف Frame Pointer) نیز مسیر قابل‌دیباگ را محو می‌کند. [۲۴، ۲۵]

۳. **کرش/بی‌ثباتی در دیباگ چندریسمانی (Multi-Thread)**

وقفهٔ نرم‌افزاری (INT3) زمان‌بندی نخ‌ها را تغییر می‌دهد؛ گاه باعث محوشدن یا ایجاد Race می‌شود. اعمال Breakpoint سراسری روی همهٔ نخ‌ها ممکن است Deadlock، توقف ناخواسته یا حتی کرش ابزار را رقم بزند. هماهنگ‌سازی step-over/into بین نخ‌ها نیز در برخی ابزارها قابل اتکا نیست. [۲۲، ۲۴]

۴. **محدودیت انواع** Breakpoint

رجیسترهای سخت‌افزاری (DR0–DR3) تعداد محدودی Watchpoint می‌دهند؛ Breakpoint نرم‌افزاری بایت کد را تغییر می‌دهد و توسط ضددیباگ‌ها شناسایی می‌شود؛ Page-Guard پوشش کامل ندارد و سربار ایجاد می‌کند. [۲۲، ۲۶]

۵. **قیود سطح کرنل و وابستگی به پلتفرم**

دیباگ Kernel-Mode نیازمند دسترسی‌های ویژه، Secure-Boot سازگار و سمبل‌های نسخه‌همخوان است؛ کوچک‌ترین عدم همخوانی، تحلیل Stack و Context را بی‌اعتبار می‌کند. [۲۷، ۲۳]

تکنیک‌های ضددیباگ (Anti-Debugging) و مشکلات امنیتی:

۱. **تشخیص دیباگر**

APIهای رایج: `IsDebuggerPresent`، `CheckRemoteDebuggerPresent`، `NtQueryInformationProcess` (پرچم‌های ProcessDebugPort/ProcessDebugFlags)، بررسی رجیسترهای DRx. بررسی مصنوعات: حضور پنجره/فرآیند شناخته‌شده (x64dbg، IDA)، وجود هندل‌های مشکوک، نام‌گذاری Pipe/Mutex خاص. چک یکپارچگی بایت‌ها برای شناسایی INT3. [۲۲، ۲۸، ۲۶]

۲. **آنتی-Attach و آنتی-Hook**

استفاده از `CreateProcess` با فلگ‌های خاص، محافظت از نخ‌ها، تغییر ACL روی اشیاء کرنلی، یا اختصاص‌دادن استثناها به هندلر داخلی (SEH) برای بی‌اثر کردن وقفه‌های دیباگر. [۲۲، ۲۴، ۲۷]

۳. **آنتی-Single-Step و زمان‌سنجی**

بررسی Trap Flag، استفاده از QueryPerformanceCounter/RDTSC برای کشف تأخیر stepping، اندازه‌گیری شکست‌های کش و شاخه‌ها. برخی برنامه‌ها با شناسایی کمترین انحراف زمانی، مسیر بدیل/بن‌بست اجرا می‌کنند. [۲۲، ۲۶، ۲۹]

۴. **کد خود-تغییرده/پک‌شده (Self-modifying / Packed)**

آنپک/دیکریپت در زمان اجرا باعث می‌شود break روی بخش‌های «قبل از گسترش کد» بی‌اثر باشد. تغییر مداوم نقشهٔ حافظه، آدرس‌ها و CRC-checkها ممانعت اضافی ایجاد می‌کند. [۲۲، ۳۰]

۵. **سوءاستفاده از ویژگی‌های زبان/سیستم**

TLS Callbacks، اتصالات تأخیری (delay-load)، inline syscalls، و تکنیک‌های SEH-Obfuscation مسیر کنترل را پنهان می‌کنند و بازسازی جریان (Control-Flow) را دشوار می‌سازند. [۲۴، ۳۰]

۶. **ریسک‌های امنیتی محیط تحلیل**

اجرای نمونه‌های ناشناس (به‌خصوص بدافزار) در دیباگر ممکن است به فرار از VM، آلودگی میزبان یا افشای شبکه منجر شود؛ پیکربندی نادرست snapshot/ایزولیشن و پوشه‌های اشتراکی ریسک را بالا می‌برد. [۲۲، ۲۴]

محدودیت‌های قانونی یا اخلاقی در استفاده از ابزارهای خاص:

مجوز و حق مؤلف – مهندسی معکوس برای سازگاری/آموزش در برخی حوزه‌ها معافیت دارد، ولی دور زدن سازوکارهای حفاظت (Obfuscation/DRM) در بسیاری نظام‌های حقوقی می‌تواند نقض قانون باشد. [۳۰]

مالکیت داده و حریم خصوصی – دیباگ روی داده‌های کاربران/شرکت‌ها باید با مجوز صریح، محیط ایزوله و سیاست نگهداشت انجام شود. [۲۲]

ایمنی آزمایشگاه – تحلیل بدافزار یا ابزارهای با کارکرد دوگانه بدون رویه‌های استاندارد (شبکهٔ بسته، no-internet، snapshots، عدم استفاده از حساب‌های اصلی) غیرمسئولانه است. [۲۲]

افشای مسئولانه – کشف آسیب‌پذیری در حین دیباگ باید طبق خطوط‌مشی افشای مسئولانه و هماهنگی با ذی‌نفعان انجام شود. [۲۴]

فصل ۸

نتیجه‌گیری و پیشنهادات آینده

۱.۸ مرور کلی یافته‌ها

در این گزارش، سیر تحول مهندسی نرم‌افزار از روش‌های ابتدایی و فاقد ساختار مانند «کد و فیکس»، به مدل‌های ساختارمند و خطی نظیر «آبشاری»، و در ادامه به رویکردهای تکرار شونده و تکاملی مانند مدل «افزایشی»، «مارپیچی» و در نهایت متدولوژی‌های «چابک» (Agile) و DevOps مورد بررسی قرار گرفت. مشخص شد که هدف اصلی این تکامل، تبدیل توسعه نرم‌افزار از یک فعالیت تجربی به فرآیندی نظام‌مند برای مدیریت پیچیدگی و تولید سیستم‌های نرم‌افزاری قابل اعتماد و با کیفیت بالا بوده است. در فصل دوم، چالش‌های کلیدی در چرخه توسعه و تکامل نرم‌افزار شناسایی شدند. این مشکلات در سه دسته‌ی اصلی طبقه‌بندی شدند:

۱. **مشکلات سازمانی:** شامل ارتباطات ناکارآمد بین تیم‌ها ، مستندسازی ضعیف و مدیریت ناکارآمد تغییرات.

۲. **مشکلات فنی:** شامل انباشت بدهی فنی (Technical Debt) ، ناسازگاری با فناوری‌های جدید و چالش‌های ناشی از سیستم‌های قدیمی (Legacy Systems).

۳. **مشکلات انسانی (شامل فرسودگی تیم و فقدان مهارت‌های جدید):** مطالعات موردی پروژه‌های شکست‌خورده مانند LASCAD و File Case Virtual (VCF) نشان داد که عواملی چون تست ناکافی ، طراحی ضعیف و مدیریت تغییرات کنترل‌نشده نقش مستقیمی در شکست پروژه‌های بزرگ داشته‌اند.

در فصل سوم، DevOps به‌عنوان یک فرهنگ ، فلسفه و مجموعه‌ای از ابزارها معرفی شد که با

هدف یکپارچه‌سازی تیم‌های توسعه (Dev) و عملیات (Ops) و رفع شکاف میان آن‌ها پدید آمده است. نشان داده شد که DevOps با تکیه بر خودکارسازی فرآیندهای CI/CD و استفاده از ابزارهایی نظیر Docker ، Kubernetes و Jenkins ، منجر به افزایش سرعت تحویل نرم‌افزار ، بهبود پایداری استقرارها و ارتقای کیفیت محصول می‌شود. با این حال، چالش‌هایی نظیر مقاومت فرهنگی و پیچیدگی فنی ابزارها نیز در مسیر استقرار آن وجود دارد.

در نهایت، فصل چهارم به ضرورت باز طراحی (Reengineering) در چرخه عمر نرم‌افزار پرداخت. دلایل اصلی این نیاز، مواردی چون ضعف معماری اولیه (مانند الگوی ضد طراحی «توپ گلی بزرگ» (BBoM) ، منسوخ شدن فناوری‌ها و انباشت بدهی فنی است ؛ بدهی فنی می‌تواند هزینه‌ی فرصت سنگینی ایجاد کند، به‌طوری که توسعه‌دهندگان حدود ۴۲٪ از هفته کاری خود را صرف رسیدگی به آن می‌کنند. تکنیک‌هایی مانند بازآرایی (Refactoring) ، مهندسی معکوس (Reverse Engineering) و مهاجرت (Migration) به‌عنوان ابزارهای کلیدی معرفی شدند. همچنین مشخص شد که استفاده از ابزارهای نوین هوش مصنوعی می‌تواند فرآیند مهندسی معکوس سیستم‌های قدیمی را تسریع بخشد. در اجرای بازطراحی، استراتژی‌های مهاجرت افزایشی (مانند «الگوی انجیر خفه‌کننده») ریسک بسیار کمتری نسبت به رویکرد «انفجار بزرگ» دارند. مطالعات موردی نشان داد که بازطراحی می‌تواند اهداف استراتژیک متفاوتی داشته باشد؛ از بهبود تجربه کاربری و شخصی‌سازی (مانند اپلیکیشن PayPal) تا یکپارچه‌سازی خدمات بانکی و ابزارهای عملیاتی برای پاسخ به نیازهای بازار (مانند نئو بانک فوربیکس). تصمیم‌گیری نهایی برای باز طراحی نیازمند ارزیابی دقیق معیارهایی چون هزینه ، زمان ، ریسک و تاثیر بر کیفیت است.

۲.۸ تأثیر DevOps و بازطراحی بر پایداری نرم‌افزار

پایداری نرم‌افزار، به معنای توانایی سیستم برای ادامه عملکرد صحیح و قابلیت تکامل در طول زمان، به شدت تحت تأثیر چالش‌های نگهداری است. بخش عمده‌ای از هزینه‌های چرخه عمر نرم‌افزار (حدود ۶۰ تا ۸۰ درصد) صرف نگهداری و تکامل می‌شود. همچنین، هزینه‌های پنهان ناشی از بدهی فنی، پایداری بلندمدت پروژه‌ها را تهدید می‌کند.

رویکردهای DevOps و بازطراحی، راهکارهای مستقیمی برای افزایش پایداری و کاهش این هزینه‌ها ارائه می‌دهند. DevOps با خودکارسازی فرآیندهای تست و ادغام (CI) ، باعث کشف سریع خطاها در مراحل اولیه توسعه می‌شود. این امر هزینه‌های نگهداری بلندمدت را به طور قابل توجهی کاهش می‌دهد. فرهنگ DevOps مبتنی بر چرخه‌های بازخورد سریع و مسئولیت مشترک است ؛ رویکردی مانند "You build، run you it" در آمازون تضمین می‌کند که توسعه‌دهندگان به پایداری محصول

پس از استقرار نیز متعهد باشند ، که این امر مستقیماً به ارتقای کیفیت و پایداری سیستم کمک می‌کند.

از سوی دیگر، بازطراحی برای پایداری سیستم‌های قدیمی (Legacy Systems) که اغلب پیچیده و فاقد مستندات هستند، حیاتی است. تکنیک‌هایی مانند Refactoring با بهبود ساختار داخلی کد و افزایش خوانایی، قابلیت نگهداری سیستم را افزایش داده و هزینه اصلاح بدهی فنی را در بلند مدت کاهش می‌دهد. مهاجرت (Migration) نیز به سازمان‌ها اجازه می‌دهد تا از فناوری‌های منسوخ که دارای ریسک‌های امنیتی و عملیاتی هستند، فاصله بگیرند.

در مجموع، DevOps فرآیند تکامل و تحول مداوم را ممکن می‌سازد، در حالی که بازطراحی تضمین می‌کند که پایه‌های فنی سیستم برای این تکامل مستمر، مستحکم و قابل نگهداری باقی بمانند.

۳.۸ توصیه‌ها برای تیم‌های مهندسی نرم‌افزار

بر اساس یافته‌های این گزارش و چالش‌های مطرح شده در فصول قبل، توصیه‌های زیر برای تیم‌های مهندسی نرم‌افزار جهت بهبود فرآیندهای توسعه، نگهداری و تکامل نرم‌افزار ارائه می‌گردد:

۱. **پذیرش فرهنگ DevOps فراتر از ابزارها:** به یاد داشته باشید که DevOps پیش از هر چیز یک تغییر فرهنگی است. بر شکستن سیلوها (تیم های ایزوله)، ایجاد ارتباط ارتباط باز، همکاری میان‌تیمی و مسئولیت مشترک تمرکز کنید تا مقاومت‌های فرهنگی کاهش یابد.

۲. **بازطراحی به عنوان یک فرصت استراتژیک:** بازطراحی را نه فقط یک رفع نقص فنی، بلکه یک فرصت استراتژیک برای بازنگری در مدل کسب‌وکار (مانند فوربیکس) یا بهبود چشمگیر تجربه کاربری (مانند پی‌پال) در نظر بگیرید.

۳. **مدیریت فعال بدهی فنی:** بدهی فنی را به‌عنوان بخشی از فرآیند بپذیرید اما برای بازپرداخت آن برنامه‌ریزی منظم داشته باشید. برای توجیه اقتصادی بازطراحی، از معیارهای کمی مانند «نسبت بدهی فنی» (TDR) و محاسبه‌ی هزینه‌ی فرصت ناشی از زمان صرف شده بر روی بدهی فنی (حدود ۴۲٪) استفاده کنید.

۴. **اجرای فرآیندهای مدیریت تغییر (change Management):** برای جلوگیری از آشفتگی و ناسازگاری‌های ناشی از تغییرات کنترل‌نشده (که عامل شکست پروژه‌هایی چون LASCAD بود)، فرآیندهای

رسمی مدیریت تغییر را پیاده‌سازی کنید. این فرآیند باید شامل مستندسازی تغییرات، تحلیل تاثیر و تست پیش از اجرا باشد.

۵. **سرمایه‌گذاری بر خودکارسازی(CI/CD):** از ابزارهای ادغام مداوم (CI) و تحویل مداوم (CD) مانند Jenkins یا CI/CD GitLab برای خودکارسازی ساخت، تست و استقرار استفاده کنید. این کار به کشف سریع خطاها و افزایش پایداری استقرارها کمک شایانی می‌کند.

۶. **برنامه‌ریزی استراتژیک برای سیستم‌های قدیمی :(Legacy)**

- **مهاجرت افزایشی:** در مواجهه با سیستم‌های حیاتی، از استراتژی‌های پرخطر «انفجار بزرگ» پرهیز کرده و از «الگوی انجیر خفه‌کننده» Fig (Strangler Pattern) برای مهاجرت افزایشی و کم‌ریسک استفاده کنید.
- **مهندسی معکوس هوشمند:** برای درک ساختار سیستم‌های قدیمی فاقد مستندات ، از ابزارهای مدرن هوش مصنوعی به عنوان «ابزار باستان‌شناسی» برای تسریع فرآیند تحلیل کد و مهندسی معکوس بهره ببرید.

۷. **توجه به عوامل انسانی و آموزش مستمر:** موفقیت پروژه به عوامل انسانی نیز وابسته است. با ایجاد فرهنگ کاری سالم از فرسودگی تیم جلوگیری کنید. با سرمایه‌گذاری بر آموزش و اشتراک دانش، مهارت‌های مورد نیاز برای فناوری‌های نوین را در تیم تقویت نمایید.

۸. **مستندسازی به عنوان یک دارایی کلیدی:** با مستندسازی ضعیف که فرآیندهای نگهداری و ورود اعضای جدید به تیم را مختل می‌کند، مقابله کنید. مستندات باید به عنوان منبع حیاتی دانش سازمان تلقی شده و شامل مستندات سیستم، فرآیند و تصمیمات طراحی باشند.

۴.۸ مسیرهای تحقیقاتی و آموزشی آینده

با توجه به چالش‌ها و روندهای بررسی‌شده، به‌ویژه در فصول ۳ و ۴، مسیرهای زیر برای تحقیقات و آموزش‌های آتی در حوزه مهندسی نرم‌افزار پیشنهاد می‌شود:

۱. **امنیت در چرخه‌های خودکار(DevSecOps):** همان‌طور که در چالش‌های استقرار DevOps اشاره شد، افزایش سرعت استقرار می تواند نگرانی‌های امنیتی ایجاد کند. تحقیقات و آموزش‌های آینده باید بر ادغام یکپارچه امنیت در تمام مراحل چرخه عمر نرم افزار (رویکرد DevSecOps) و روش‌های مدیریت خودکار دسترسی‌ها و داده‌های حساس متمرکز شوند.

۲. **کاربرد هوش مصنوعی در بازمهندسی نرم‌افزار:** علاوه بر استفاده‌ی فعلی از AI در درک کد، مسیرهای تحقیقاتی آینده باید بر توسعه و ارزیابی مدل‌های هوش مصنوعی برای خودکارسازی فرآیند مهندسی معکوس، استخراج منطق تجاری از کدهای قدیمی ، شناسایی الگوهای ضد طراحی و تولید خودکار مستندات فنی برای سیستم‌های Legacy متمرکز شوند.

۳. **مدیریت پیچیدگی ابزارها و زیرساخت‌های DevOps:** یکی از موانع استقرار ،DevOps پیچیدگی فنی ابزارهایی مانند Kubernetes و Terraform و بار آموزشی سنگین آن‌هاست. تحقیقات آتی می‌تواند بر **«ساده‌سازی تعامل»** با این ابزارها متمرکز باشد؛ چه از طریق توسعه‌ی **پلتفرم‌های سطح بالاتر (PaaS)** که به عنوان یک لایه‌ی انتزاعی عمل کرده و پیچیدگی‌های زیرساخت را از توسعه‌دهنده پنهان می‌کنند، و چه از طریق ایجاد **روش‌های مدیریتی هوشمندتر** و ابزارهای کمکی برای مدیریت بهینه‌ی خود این زیرساخت‌های پیچیده.

۴. **توسعه‌ی چارچوب‌های آموزشی و تحقیقاتی برای جنبه‌های انسانی DevOps:** با توجه به اینکه «مقاومت فرهنگی» یکی از مهم‌ترین موانع در استقرار موفق DevOps شناسایی شده است، یک خلاء تحقیقاتی و آموزشی آشکار وجود دارد. برنامه‌های آموزشی فعلی اغلب بیش از حد بر ابزارها متمرکز هستند. لذا، مسیرهای تحقیقاتی آینده باید بر توسعه و ارزیابی «مدل‌های مدیریت تغییر» و «تکنیک‌های روانشناسی سازمانی» متمرکز شوند که گذار فرهنگی به DevOps را تسهیل می‌کنند. همچنین، مسیرهای آموزشی آینده باید چارچوب‌هایی مدون برای آموزش مهارت‌های نرم (Soft Skills)، مانند ایجاد فرهنگ گزارش‌دهی بدون سرزنش (Blameless Postmortem) و همکاری بین‌تیمی ، در کنار آموزش‌های فنی ارائه دهند.

۵. **الگوهای پیشرفته بازطراحی و مدیریت داده در مهاجرت:** با توجه به اهمیت حیاتی سیستم‌های قدیمی (مانند سیستم‌های Core Banking) ، نیاز به الگوها و استراتژی‌های اثبات‌شده برای مدرن‌سازی آن‌ها وجود دارد. تحقیقات آینده می‌تواند بر توسعه‌ی مدل‌هایی برای ارزیابی دقیق ریسک، هزینه و زمان در سناریوهای مختلف بازطراحی ، و همچنین تحقیق بر روی الگوهای مدیریت سازگاری داده‌ها (مانند Saga و Idempotency) در طول مهاجرت افزایشی به معماری میکروسرویس تمرکز کند.

۶. **بهبود فرآیندها مبتنی بر داده‌های مانیتورینگ (AIOps):** با گسترش ابزارهای نظارت و بازخورد مانند Prometheus و Sentry ، فرصت‌های جدیدی برای استفاده از هوش مصنوعی و تحلیل داده‌های عملیاتی جهت بهبود مستمر فرآیندهای توسعه و تصمیم‌گیری‌های مبتنی بر شاخص‌های کمی (مانند SLOs) فراهم آمده است که نیازمند تحقیق و توسعه‌ی بیشتر است.

منابع

[۱] Software development history: From waterfall to Agile to DevOps. ۲۰۲۲.

[۲] “Software Development Life Cycle (SDLC)”. In: (۲۰۲۵).

[۳] Roger S. Pressman and Bruce R. Maxim. Software Engineering: A Practitioner’s Approach. McGraw-Hill Education, ۲۰۲۰.

[۴] Saurabh Jha, K. Kumar, and S. Kumar. “From theory to practice: Understanding DevOps culture and mindset”. In: Cogent Engineering ۱۰.۱ (۲۰۲۳), p. ۲۲۵۱۷۵۸. DOI: 10.1080/23311916.2023.2251758. URL: https://doi.org/10.1080/23311916.2023.2251758.

[۵] National Research Council. Intellectual Property Issues in Software. Washington, D.C.: National Academies Press, ۱۹۹۱.

[۶] TTC Consultants. Reverse Engineering and Intellectual Property Rights: Balancing Innovation and Legal Considerations. ۲۰۲۴.

[۷] Quarkslab. Reverse Engineering: A Threat to Intellectual Property of Innovations? ۲۰۲۳.

[۸] IP Watchdog. Reverse Engineering Law: Understanding Restrictions and Minimize Risks. ۲۰۲۱.

[۹] RecordPoint. The Hidden Costs of Maintaining Legacy Systems. ۲۰۲۴.

[۱۰] vFunction. How Much Does it Cost to Maintain Legacy Software Systems? ۲۰۲۲.

[۱۱] ModLogix. Legacy Software Re-engineering: Risks and Mitigations. ۲۰۲۲.

[۱۲] DevSquad. ۷ Costs of Maintaining Legacy Systems & How to Avoid Them. ۲۰۲۵.

[۱۳] Medium. When Legacy Code Becomes the Product: Lessons in Technical Debt and Missed Opportunities. ۲۰۲۳.

[۱۴] ModelCode AI Blog. The Biggest Risks of Misinterpreting Business Logic During Legacy Modernization. ۲۰۲۴.

[۱۵] JoIV Journal. Overview of Software Re-Engineering Concepts, Models and Approaches. ۲۰۲۳.

[۱۶] Integrity۳۶۰. How Legacy Software and Hardware is a Ticking Cyber Security Risk Time-bomb. ۲۰۲۳.

[۱۷] HeroDevs. How Outdated Systems and Legacy Software are Fueling Modern Cyber Attacks. ۲۰۲۴.

[۱۸] Atiba. Vulnerabilities in Using Legacy Software: Key Risks and Mitigations. ۲۰۲۵.

[۱۹] N. Dissanayake et al. “Software Security Patch Management — A Systematic Literature Review of Challenges, Approaches, Tools and Practices”. In: arXiv preprint (۲۰۲۰). eprint: 2012.00544.

[۲۰] M. Moraga and Y. Zhao. “Reverse engineering a legacy software in a complex system: A systems engineering approach”. In: INCOSE International Symposium. Vol. ۲۸. ۱. ۲۰۱۸, pp. ۱۲۵۰–۱۲۶۴.

[۲۱] A. Pascal et al. “Case Studies in Model-Driven Reverse Engineering”. In: Proceedings of the ۱۱th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC۳K). Vol. ۲. ۲۰۱۹, pp. ۷۳۱–۷۴۰.

[۲۲] Michael Sikorski and Andrew Honig. Practical Malware Analysis. No Starch Press, ۲۰۱۲.

[۲۳] IDA Pro Debugger and Graph View (Performance on Large Binaries). Online manual. Hex-Rays. ۲۰۲۰.

[۲۴] Bruce Dang et al. Practical Reverse Engineering: x۸۶, x۶۴, ARM, Windows Kernel, Reversing Tools, and Obfuscation. Wiley, ۲۰۱۴.

[۲۵] Eldad Eagle. Reversing: Secrets of Reverse Engineering. McGraw-Hill, ۲۰۰۳.

[۲۶] Intel ۶۴ and IA-۳۲ Architectures Software Developer’s Manual: Debug Registers, RDTSC, Trap Flag. Volume ۳, Online manual. Intel. ۲۰۱۹.

[۲۷] Windows Debugging with WinDbg (Symbols, Kernel Debugging). Online manual. Microsoft Docs. ۲۰۲۰.

[۲۸] x۶۴dbg Documentation and Wiki (Anti-Anti-Debug/ScyllaHide). Online manual. x۶۴dbg Project. ۲۰۲۰.

[۲۹] QueryPerformanceCounter and Timer Resolution, ETW/Perf. Online manual. Microsoft Docs. ۲۰۱۹.

[۳۰] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Tech. rep. University of Auckland, ۱۹۹۷.