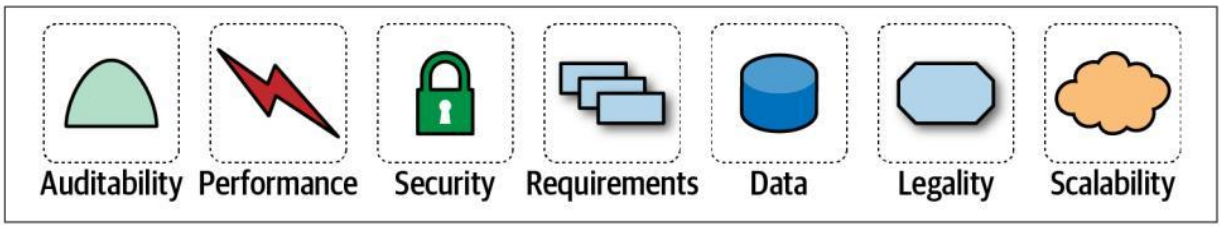
فصل ۴: تعریف ویژگی‌های معماری

وقتی شرکتی تصمیم می‌گیرد مسئله‌ای را با نرم‌افزار حل کند، فهرستی از نیازمندی‌ها را برای آن سیستم گردآوری می‌کند. تکنیک‌های بسیار متنوعی برای فرآیند گردآوری نیازمندی‌ها وجود دارد که عموماً توسط تیم تعریف می‌شوند. معمار باید عوامل بسیاری را در طراحی یک راه‌حل نرم‌افزاری در نظر بگیرد.



شکل ۴-۱. یک راه‌حل نرم‌افزاری هم از نیازمندی‌های دامنه و هم از ویژگی‌های معماری تشکیل شده است

معماران ممکن است در تعریف نیازمندی‌های کسب‌وکار همکاری کنند، اما یکی از مسئولیت‌های کلیدی آن‌ها شامل تعریف، کشف و تحلیل تمام کارهایی است که نرم‌افزار باید انجام دهد و مستقیماً به عملکرد کسب و کار مربوط نمی‌شود.

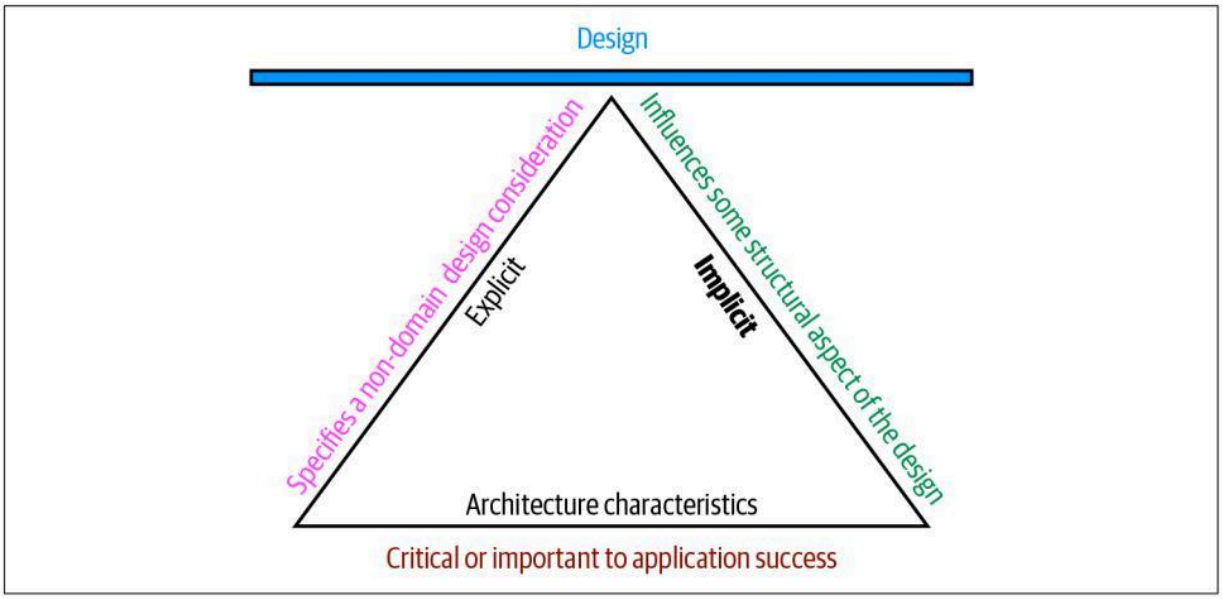
چه چیزی معماری نرم‌افزار را از برنامه نویسی متمایز می‌کند؟

موارد زیادی، از جمله نقشی که معماران در تعریف ویژگی‌های معماری دارند؛ یعنی جنبه‌های مهم سیستم که مستقل از کسب و کار هستند.

یک ویژگی معماری سه معیار را برآورده می‌کند:

- **تعریف‌کننده یک نیازمندی غیرعملکردی (Non-Functional) است:** این ویژگی مستقیماً به منطق کسب‌وکار (Domain Logic) ارتباطی ندارد. به عبارت دیگر، به جای تمرکز بر قابلیت‌های تجاری (مانند "ثبت‌نام کاربر" یا "محاسبه فاکتور")، بر **کیفیت** و **شرایط** اجرای آن قابلیت‌ها تمرکز دارد (مانند "سیستم باید سریع باشد" یا "سیستم باید امن باشد").
- **دارای تأثیر ساختاری و گسترده است:** این ویژگی یک تصمیم کوچک و محلی نیست. انتخاب آن، شالوده و اسکلت اصلی سیستم را شکل می‌دهد و بر انتخاب تکنولوژی‌ها، الگوهای طراحی، و نحوه ارتباط اجزای مختلف سیستم تأثیر عمیقی می‌گذارد. نمی‌توان آن را به سادگی در انتهای پروژه "اضافه" کرد.
- **برای موفقیت کسب‌وکار، حیاتی و غیرقابل چشم‌پوشی است:** اگر این ویژگی به درستی پیاده‌سازی نشود، کل برنامه، حتی با وجود عملکرد صحیح منطق تجاری، با شکست مواجه خواهد شد. این ویژگی مستقیماً به اهداف کسب‌وکار گره خورده است.

این بخش‌های درهم‌تنیده تعریف ما در شکل ۴-۲ نشان داده شده است.



شکل ۴-۲. ویژگی‌های متمایزکننده مشخصه‌های معماری

برای موفقیت برنامه، حیاتی یا مهم است

برنامه‌ها *می‌توانند* از تعداد زیادی ویژگی معماری پشتیبانی کنند... اما نباید. پشتیبانی از هر ویژگی معماری به پیچیدگی طراحی می‌افزاید. بنابراین، یک کار حیاتی برای معماران، انتخاب کمترین تعداد ویژگی‌های معماری به جای بیشترین تعداد ممکن است.

در معماری نرم‌افزار، ما با دو دسته ویژگی روبرو هستیم: **ویژگی‌های صریح** که به وضوح در مستندات درخواست شده‌اند، و **ویژگی‌های ضمنی** که حیاتی هستند اما به دلیل بدیهی بودن در آن حوزه، ممکن است هرگز نوشته نشوند (مانند سرعت بالا در سیستم‌های مالی).

این وظیفه معمار است که با درک عمیق از کسب‌وکار، این نیازمندی‌های پنهان را شناسایی و در طراحی لحاظ کند.

نکته مهم این است که این ویژگی‌ها اغلب در تقابل با یکدیگر قرار دارند. به همین دلیل، معماران دائماً در حال **"بده‌بستان"** هستند؛ یعنی باید تصمیم بگیرند که برای به دست آوردن یک مزیت (مثلاً امنیت بیشتر)، کدام ویژگی دیگر (مثلاً سرعت پاسخ‌دهی) را فدا کنند تا به بهترین تعادل ممکن برای موفقیت سیستم برسند.

فهرست (جزئی) ویژگی‌های معماری

ویژگی‌های معماری (از سطح کد تا عملیات) فاقد استاندارد جهانی هستند و تعاریفشان با تکامل تکنولوژی دائماً تغییر می‌کند. برای مدیریت این پیچیدگی، معماران این ویژگی‌ها را در دسته‌بندی‌های کلی سازماندهی می‌کنند تا تحلیل آن‌ها ساده‌تر شود.

Operational Architecture Characteristics

ویژگی‌های معماری که به جنبه‌های عملیاتی سیستم می‌پردازند، مرز مشترک بین دنیای معماری و دنیای DevOps هستند.

۱. دسترس‌پذیری (Availability)

- **تعریف ساده:** سیستم چقدر "زنده" و قابل استفاده است؟
- **توضیح کامل:** دسترس‌پذیری با "تعداد نُه" (Nines) سنجیده می‌شود. مثلاً "پنج تا نه" (99.999%) یعنی سیستم در طول یک سال، تنها حدود ۵ دقیقه قطعی خواهد داشت.
- **قیاس:** تابلو "باز است" (Open) یک فروشگاه که در ساعات کاری همیشه روشن باشد.
- **مثال فنی:** استفاده از **Load Balancer** که ترافیک را بین چندین سرور پخش می‌کند.
- **بده‌بستان (Trade-off):** دسترس‌پذیری بالا بسیار **پرهزینه** است و نیاز به سخت‌افزار اضافی (Redundancy) دارد.

۲. تداوم (Continuity) و قابلیت بازیابی (Recoverability)

- **تعریف ساده:** اگر یک **فاجعه** رخ دهد، چقدر سریع می‌توانیم سیستم را برگردانیم و چقدر داده از دست می‌دهیم؟
- **توضیح کامل:** این ویژگی با فجایع بزرگ سروکار دارد و با دو معیار سنجیده می‌شود:
  - **RTO (Recovery Time Objective):** حداکثر زمان مجاز برای آفلاین ماندن.
  - **RPO (Recovery Point Objective):** حداکثر حجم داده‌ای که از دست می‌رود.
- **قیاس:** اگر انبار اصلی فروشگاه آتش بگیرد، چقدر سریع می‌توان از انبار پشتیبان کار را ادامه داد (RTO) و سفارشات چند ساعت آخر از بین می‌روند (RPO).
- **مثال فنی:** داشتن یک نسخه پشتیبان کامل در یک منطقه جغرافیایی متفاوت.
- **بده‌بستان:** این گران‌ترین ویژگی معماری است و هزینه‌ها را می‌تواند دو برابر کند.

۳. عملکرد (Performance)

- **تعریف ساده:** سیستم چقدر **سریع** و **کارآمد** است؟
- **توضیح کامل:** عملکرد شامل **زمان پاسخ (Latency)**، **توان عملیاتی (Throughput)**، و **عملکرد تحت بار (Load)** است.
- **قیاس:** رستورانی که هم سریع سفارش آماده می‌کند (Latency)، هم در ساعت تعداد زیادی سفارش تحویل می‌دهد (Throughput)، و هم در شب‌های شلوغ کیفیتش افت نمی‌کند (Load).
- **مثال فنی:** استفاده از **کش (Caching)** و بهینه‌سازی کوئری‌های دیتابیس.
- **بده‌بستان:** عملکرد بالا ممکن است با **ثبات داده‌ها (Consistency)** در تضاد باشد.

۴. پایایی/ایمنی (Reliability/Safety)

- **تعریف ساده:** آیا سیستم کار درست را به طور مداوم انجام می‌دهد؟ و اگر شکست بخورد، آیا به شکل امنی شکست می‌خورد؟
- **توضیح کامل: پایایی (Reliability)** به صحت عملکرد اشاره دارد (مانند سیستم حسابداری دقیق). **ایمنی (Safety)** مربوط به سیستم‌هایی است که خرابی آن‌ها فاجعه‌بار است و باید به حالت امن وارد شوند (Fail-Safe).
- **قیاس:** پایایی مانند ماشین حسابی است که همیشه جواب درست می‌دهد. ایمنی مانند ترمز اضطراری آسانسور است.
- **مثال فنی:** نرم‌افزار کنترل پرواز هواپیما باید ایمنی بالایی داشته باشد.
- **بده‌بستان:** دستیابی به این دو، نیازمند تست‌های سخت‌گیرانه است که **سرعت توسعه** را کاهش می‌دهد.

۵. استواری (Robustness)

- **تعریف ساده:** سیستم در برابر ورودی‌های نامعتبر، خطاهای غیرمنتظره و شرایط سخت چگونه رفتار می‌کند؟
- **توضیح کامل:** استواری یعنی توانایی مدیریت خطا بدون از کار افتادن (Crash).
- **قیاس:** یک راننده حرفه‌ای که در شرایط سخت و غیرمنتظره می‌تواند ماشین را کنترل کند.
- **مثال فنی:** استفاده از بلوک‌های try-catch و الگوی **Circuit Breaker**.
- **بده‌بستان:** نوشتن کد استوار، **پیچیدگی کد** را افزایش می‌دهد.

۶. مقیاس‌پذیری (Scalability)

- **تعریف ساده:** آیا سیستم می‌تواند با افزایش بار (کاربران، داده‌ها) رشد کند بدون افت عملکرد؟
- **توضیح کامل:** شامل **مقیاس‌پذیری عمودی (Vertical)** (افزایش منابع یک سرور) و **افقی (Horizontal)** (اضافه کردن سرورهای بیشتر) می‌شود.

- **قیاس:** یک رستوران زنجیره‌ای که با افزایش تقاضا، شعبه‌های جدیدی افتتاح می‌کند (افقی).
- **مثال فنی:** طراحی **سرویس‌های بی‌حالت (Stateless)** و استفاده از معماری میکروسرویس.
- **بده‌بستان:** معماری‌های مقیاس‌پذیر **پیچیده‌تر** هستند و مدیریت آن‌ها دشوارتر است.

Structural Architecture Characteristics

نقش معمار تا عمق ساختار کد نیز نفوذ می‌کند و شامل مسئولیت کیفیت داخلی کدبیس می‌شود.

۱. قابلیت پیکربندی (Configurability)

- **تعریف ساده:** کاربران نهایی چقدر می‌توانند رفتار نرم‌افزار را بدون کدنویسی تغییر دهند؟
- **مثال فنی:** یک پنل تنظیمات که به مدیر اجازه می‌دهد مراحل فرآیند فروش را تعریف کند.
- **بده‌بستان:** پیاده‌سازی این قابلیت، **پیچیدگی** اولیه کد را بالا می‌برد.

۲. توسعه‌پذیری (Extensibility)

- **تعریف ساده:** چقدر راحت می‌توان قابلیت‌های جدید به سیستم اضافه کرد بدون دستکاری کدهای اصلی؟
- **مثال فنی:** سیستم **پلاگین** در نرم‌افزاری مانند وردپرس.
- **بده‌بستان:** طراحی آن نیازمند **صرف زمان و تفکر زیاد در ابتدای پروژه** است.

۳. قابلیت نصب (Installability)

- **تعریف ساده:** فرآیند نصب و راه‌اندازی اولیه نرم‌افزار چقدر سریع، ساده و بدون خطا است؟
- **مثال فنی:** ارائه یک **تصویر داکر (Docker Image)** برای نرم‌افزار.
- **بده‌بستان:** ساخت یک نصب‌کننده خوب، نیازمند **زمان و تخصص** است.

۴. قابلیت اهرم‌سازی/استفاده مجدد (Leverageability/Reuse)

- **تعریف ساده:** چقدر می‌توان از کدها یا کامپوننت‌های این پروژه در پروژه‌های آینده استفاده کرد؟
- **مثال فنی:** ساخت یک میکروسرویس مستقل برای **احراز هویت** و استفاده از آن در چندین محصول.
- **بده‌بستان:** ساخت یک کامپوننت قابل استفاده مجدد، **زمان‌برتر و پیچیده‌تر** است.

۵. محلی‌سازی (Localization - L10n)

- **تعریف ساده:** تطبیق نرم‌افزار برای استفاده در مناطق جغرافیایی و فرهنگی مختلف.
- **مثال فنی:** استفاده از کتابخانه‌های **i18n** برای پشتیبانی از زبان‌های مختلف و فرمت‌های تاریخ.
- **بده‌بستان:** نیازمند **هزینه (برای ترجمه)** و **پیچیدگی فنی** قابل توجهی است.

۶. قابلیت نگهداری (Maintainability)

- **تعریف ساده:** چقدر راحت می‌توان باگ‌ها را رفع کرد، تغییرات کوچک اعمال کرد یا کد را بهبود داد؟
- **مثال فنی:** پیروی از اصول **SOLID** و نوشتن **Unit Test**.
- **بده‌بستان:** نوشتن کد تمیز در ابتدا **زمان بیشتری** می‌برد، اما هزینه بلندمدت را کاهش می‌دهد.

۷. قابلیت حمل (Portability)

- **تعریف ساده:** چقدر راحت می‌توان نرم‌افزار را از یک محیط (مثلاً یک پایگاه‌داده) به محیطی دیگر منتقل کرد؟
- **مثال فنی:** استفاده از یک **ORM** که کد را از پایگاه‌داده خاص مستقل می‌کند.
- **بده‌بستان:** استفاده از لایه‌های انتزاعی ممکن است کمی **عملکرد** را کاهش دهد.

۸. قابلیت پشتیبانی (Supportability)

- **تعریف ساده:** وقتی مشکلی در Production رخ می‌دهد، چقدر راحت می‌توان علت آن را پیدا کرد؟
- **مثال فنی:** ثبت **لاگ‌های ساختاریافته** و استفاده از ابزارهای مانیتورینگ مانند Prometheus.
- **بده‌بستان:** نیازمند **زیرساخت و هزینه** است.

۹. قابلیت ارتقا (Upgradeability)

- **تعریف ساده:** چقدر راحت می‌توان نرم‌افزار را به نسخه جدیدتر به‌روزرسانی کرد؟
- **مثال فنی:** استفاده از استراتژی‌های استقرار مانند **Blue-Green Deployment**.
- **بده‌بستان:** نیازمند **زیرساخت پیچیده و ابزارهای DevOps پیشرفته** است.

Cross-Cutting Architecture Characteristics

این ویژگی‌ها تمام بخش‌های سیستم را تحت تأثیر قرار می‌دهند.

۱. دسترس‌پذیری (Accessibility - a11y)

- **تعریف ساده:** طراحی نرم‌افزار برای **همه**، فارغ از توانایی‌های جسمی، ذهنی یا حسی آن‌ها.
- **مثال فنی:** قراردادن متن جایگزین (alt text) برای تصاویر برای نرم‌افزارهای صفحه‌خوان.
- **بده‌بستان:** نیازمند **زمان توسعه و تست بیشتر** است.

۲. قابلیت بایگانی (Archivability)

- **تعریف ساده:** استراتژی ما برای مدیریت داده‌های قدیمی و غیرفعال چیست؟
- **مثال فنی:** انتقال خودکار داده‌های قدیمی به یک فضای ذخیره‌سازی ارزان‌تر (Cold Storage).
- **بده‌بستان: پیچیدگی** به سیستم اضافه می‌کند و بازیابی داده‌ها را کندتر می‌کند.

۳. احراز هویت (Authentication - AuthN)

- **تعریف ساده:** فرآیند تأیید هویت یک کاربر: **"شما چه کسی هستید؟"**
- **مثال فنی:** مقایسه هش رمز عبور وارد شده با مقدار ذخیره شده در دیتابیس.
- **بده‌بستان:** افزایش امنیت (مانند MFA) می‌تواند **تجربه کاربری** را پیچیده‌تر کند.

۴. مجوزدهی (Authorization - AuthZ)

- **تعریف ساده:** پس از تأیید هویت، سیستم می‌پرسد: **"شما مجاز به انجام چه کارهایی هستید؟"**
- **مثال فنی:** استفاده از **RBAC (Role-Based Access Control)** برای کنترل دسترسی بر اساس نقش کاربر.
- **بده‌بستان:** طراحی یک سیستم مجوزدهی دقیق، **پیچیدگی فنی و مدیریتی** زیادی دارد.

۵. قانونی (Legal)

- **تعریف ساده:** نرم‌افزار باید از چه قوانین، مقررات و استانداردهای صنعتی پیروی کند؟
- **مثال فنی:** پیاده‌سازی قابلیت **"حق فراموش شدن"** برای انطباق با **GDPR**.
- **بده‌بستان:** انطباق با قوانین، اغلب **محدودیت‌های فنی شدید** و **هزینه‌های بالا** را تحمیل می‌کند.

۶. حریم خصوصی (Privacy)

- **تعریف ساده:** محافظت از داده‌های حساس کاربران، حتی در برابر افراد مجاز **داخلی** شرکت.
- **مثال فنی:** رمزنگاری سرتاسری (End-to-End Encryption) در پیام‌رسان‌ها.
- **بده‌بستان: بسیار پیچیده و پرهزینه** است و می‌تواند قابلیت‌هایی مانند جستجو در سمت سرور را محدود کند.

۷. امنیت (Security)

- **تعریف ساده:** محافظت کلی سیستم و داده‌ها در برابر دسترسی، تغییر یا تخریب **غیرمجاز**.
- **مثال فنی:** استفاده از HTTPS برای رمزنگاری ارتباطات (Encryption in Transit).
- **بده‌بستان:** امنیت تقریباً همیشه با **عملکرد** و **راحتی کاربر** در تضاد است.

۸. قابلیت پشتیبانی (Supportability)

- **تعریف ساده:** زمانی که مشکلی در سیستم رخ می‌دهد، چقدر راحت می‌توانیم علت آن را پیدا کرده و آن را برطرف کنیم؟
- **مثال فنی:** استفاده از ابزارهایی مانند Datadog یا ELK Stack برای جمع‌آوری لاگ و مانیتورینگ.
- **بده‌بستان:** پیاده‌سازی زیرساخت کامل آن، نیازمند **هزینه** و **سربار عملکردی** است.

۹. قابلیت استفاده/دستیابی (Usability/Achievability)

- **تعریف ساده:** نرم‌افزار چقدر برای کاربر نهایی، **ساده، قابل فهم و لذت‌بخش** است؟
- **مثال فنی:** ارائه پیام‌های خطای واضح و کاربردی به جای کدهای خطا.
- **بده‌بستان:** نیازمند سرمایه‌گذاری قابل توجه در **تحقیق، طراحی UX/UI، و توسعه فرانت‌اند** است.

ابهامات فراوان در معماری نرم‌افزار

یکی از چالش‌های همیشگی معماران، نبود یک واژه‌نامه استاندارد و جهانی است. این وضعیت باعث سردرگمی در سطح صنعت می‌شود. بهترین راهکار، الهام از رویکرد **«طراحی دامنه-محور» (DDD)** و ایجاد یک **«زبان فراگیر» (Ubiquitous Language)** برای هر تیم است تا همه اعضا درک یکسانی از مفاهیم داشته باشند.

بده‌بستان‌ها و معماری کمترین-بدی

در دنیای واقعی، ساختن یک نرم‌افزار که همزمان فوق‌العاده امن، سریع، مقیاس‌پذیر و ارزان باشد، غیرممکن است، زیرا این ویژگی‌ها با هم تضاد دارند. این فرآیند مثل خلبانی هلیکوپتر است؛ یک تمرین مداوم برای ایجاد تعادل.

به همین دلیل، یک معمار خوب به دنبال معماری "بی‌نقص" نمی‌گردد، بلکه به دنبال معماری **"کمترین-بدی"** است؛ یعنی طرحی که بهترین تعادل ممکن را بین خواسته‌های متضاد برقرار کند.

تلاش برای راضی کردن همه، به معماری پیچیده و سنگینی منجر می‌شود که در نهایت شکست می‌خورد. راه حل چیست؟ **چابک بودن.** معماری را باید طوری طراحی کرد که بتوان آن را به مرور زمان و به صورت **تکرارشونده** بهبود داد. این کار به شما اجازه می‌دهد که با کمترین استرس شروع کنید و با یادگیری بیشتر، طراحی خود را تکامل دهید.