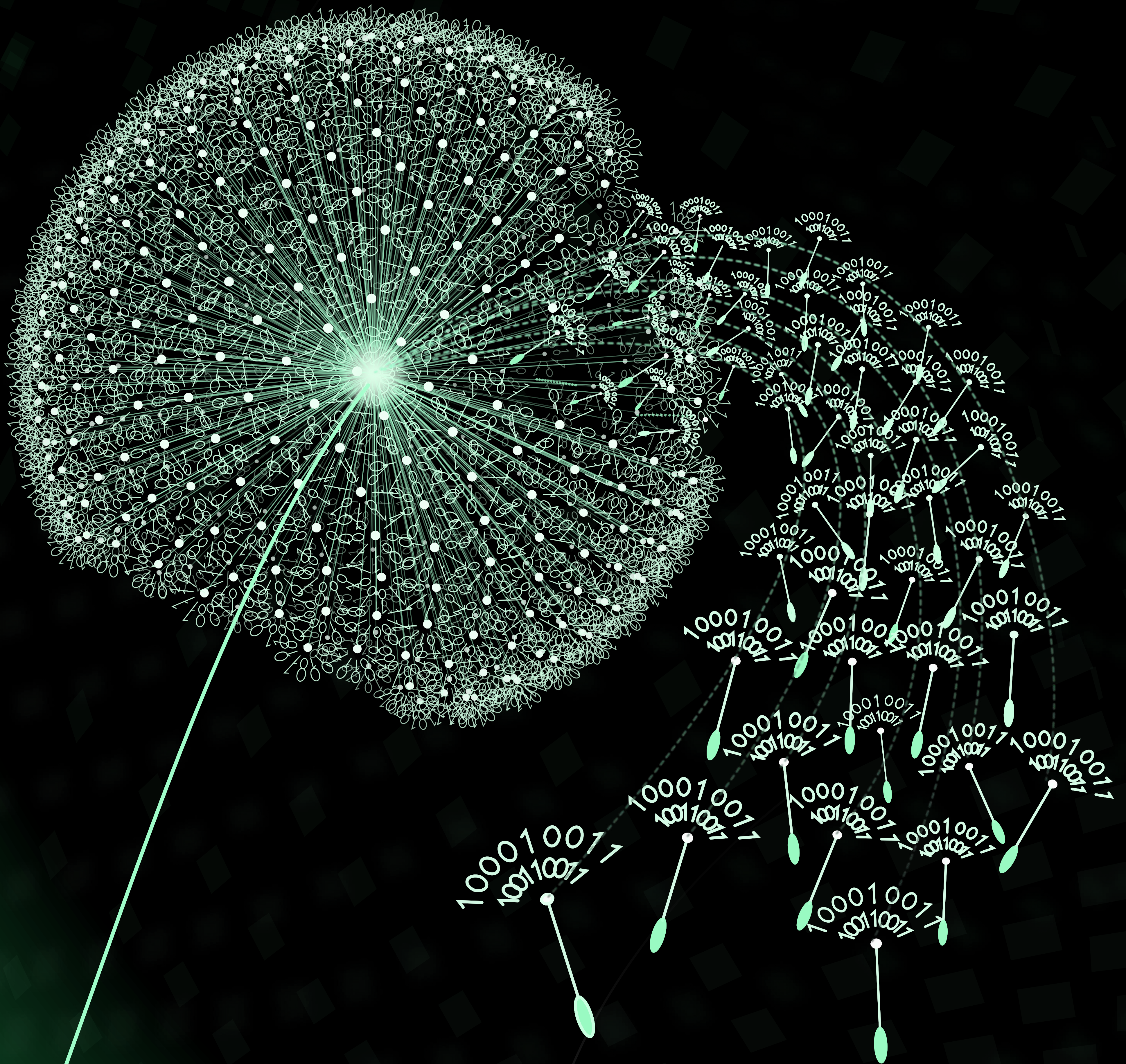


شماره دهم | مهر ۱۴۰۵ | دانشگاه صنعتی شریف

byte Magazine



انجمن علمی دانشکده مهندسی کامپیوتر
نشریه علمی فرهنگی **بایت**

چگونه میلیون ها سوکت TCP
را بدون سرخ کردن CPU هندل کنیم؟

سپهر کلانکی

نمایشگری که کندتر و
گران تر است؛ پس چرا
هنوز می خواهیمش؟

دکتر سولماز سلیمی

باهوش مردن، یا بی هوش

ماندن؟

آرش شاه حسینی

فناوری هراسی؛ امنیت یا
ناآگاهی؟ از نوکیا بپرسیم!

سعید فراتی کاشانی

پایان برنامه نویسی؟

Logan Kugler

مهندسی نرم افزار
در عصر هوش مصنوعی

مانی ابراهیمی

درف زن؛
تصمیم بگیری یا نه!

معین آعلی



صاحب امتیاز

انجمن علمی دانشکده مهندسی کامپیوتر دانشگاه صنعتی شریف

سر دبیر

آرش شاه حسینی

مدیر مسئول

امیرحسین شهیدی

صفحه آرایی و گرافیک

امیررضا جعفری - محمدپارسا آرانی - امیررضا اینانلو - نرگس کاری

تحریریه

معین آعلی - سعید فراتی کاشانی - آرش شاه حسینی - امیرمهدی نامجو - آرمان طهماسبی زاده - نیما شیرزادی

نویسندگان

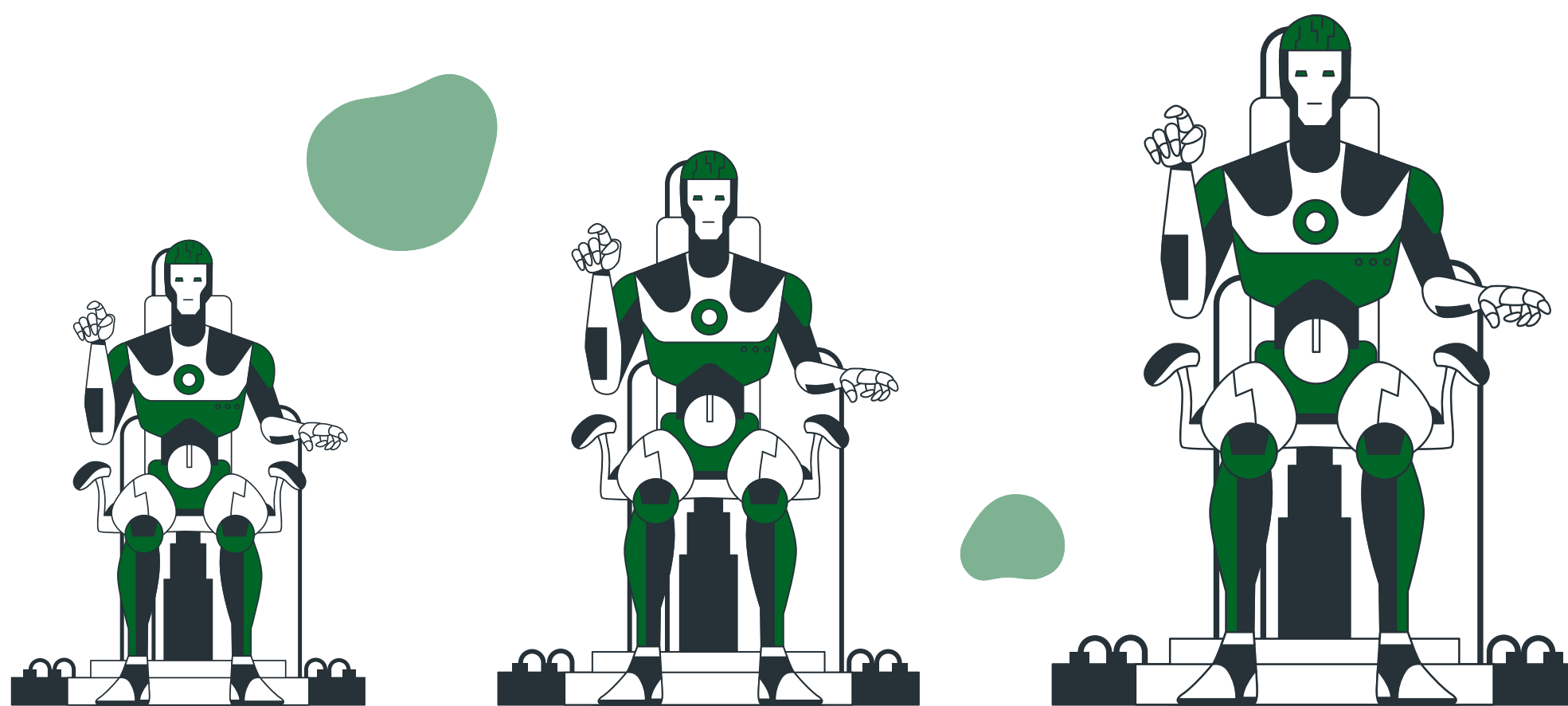
دکتر سولماز سلیمی - سپهر کلانکی - مانی ابراهیمی - Logan Kugler - سعید فراتی کاشانی - معین آعلی

ویراستاری

امیرحسین صوری

فنی

معین آعلی



سرمقاله

با هوش مردن، یا بی‌هوش ماندن؟

دانشگاه صنعتی شریف

۳ دقیقه

آرش شاه‌حسینی

کارشناسی ۱۴۰۲



روشن است که بشر ابزار می‌سازد تا زندگی را آسان‌تر کند، محدودیت‌هایش را کنار بزند و از مرز ناتوانی‌هایش فراتر برود؛ از ساخت چرخ‌ها برای جبران ناتوانی جسم‌مان، هواپیماها برای تحقق رؤیای پرواز گرفته تا پردازنده‌ها برای عبور از محدودیت‌های ذهن خطایزمان. اما امروز، شاید برای نخستین بار، با پرسشی متفاوت روبه‌رو شده‌ایم: چه می‌شود اگر ابزاری بسازیم که نه فقط سریع‌تر و قدرتمندتر، بلکه از خودمان باهوش‌تر باشد؟ از نقطه‌ای سخن می‌گوییم که می‌توان گفت اجتناب‌ناپذیر است و در عین حال بیخ گوش‌مان؛ نقطه‌ای که آن را «تکینگی فناوری»^۱ می‌نامند. وضعیتی که در آن، پیشرفت هوش مصنوعی ممکن است از توانایی ما برای درک و مهار آن پیشی بگیرد؛ تا جایی که دیگر ندانیم آن چه ساخته‌ایم، به کجا می‌رود و چه مسیری را در پیش می‌گیرد. تصور کنید سامانه‌ای بتواند نسخه‌ای توانمندتر از خود طراحی کند و آن نسخه نیز همین فرایند را تکرار کند. در چنین سناریویی، شاید فاصله میان پیشرفت‌هایی که زمانی به چند نسل نیاز داشتند، به چند سال، چند روز یا حتی کم‌تر کاهش یابد. البته هنوز نمی‌دانیم چنین آینده‌ای تا چه اندازه محتمل است، چه رسد به این که بتوانیم زمان وقوع آن را پیش‌بینی کنیم. اما اهمیت تکینگی فناوری تنها در احتمال وقوعش نیست؛ بلکه در پرسش‌هایی است که همین امروز پیش روی ما قرار می‌دهد و بهتر است برای رویارویی با آن خودمان را آماده کنیم.

ما قرن‌ها هوش و توانایی اندیشیدن را از ویژگی‌های ممتاز انسان دانسته‌ایم؛ واضح است بخش بزرگی از آن چه امروز بر زمین ساخته‌ایم، حاصل همین توانایی باشد. فرهنگ، آموزش و حتی تصور ما از ارزش کار انسانی، تا اندازه زیادی بر همین تمایز بنا شده است. حال اگر روزی سامانه‌ای بتواند بهتر از ما استدلال کند، کشف کند، بنویسد و بیافریند، چه بر سر این تمایز خواهد آمد؟ آیا باید در تعریف خلاقیت، تخصص و حتی هویت انسانی تجدیدنظر کنیم؟ آیا آن چه روزگاری هنر می‌نامیدیم و به آن افتخار می‌کردیم، در جهانی که ماشین‌ها نیز قادر به آفرینش آن هستند، همچنان همان معنا را خواهد داشت؟

شاید پرسش مهم‌تر این نباشد که ماشین‌ها تا چه اندازه به انسان شبیه خواهند شد؛ بلکه این باشد که انسان‌ها در مواجهه با ماشین‌های هوشمند، چگونه تغییر خواهند کرد.

تاریخ نشان داده است که توانایی ساختن یک فناوری، الزاماً به معنای آمادگی فرهنگی و اجتماعی برای پذیرش پیامدهای آن نیست. ما بارها ابزارهایی ساخته‌ایم که توانایی‌مان را افزایش داده‌اند، بی‌آن‌که به همان اندازه در درک مسئولیت ناشی از این قدرت پیشرفت کرده باشیم. ممکن است بتوانیم ماشین‌هایی بسازیم که بهتر از ما فکر کنند، اما هنوز باید بیاموزیم چگونه درباره استفاده از قدرت آن‌ها تصمیم بگیریم.

و شاید طنز ماجرا همین‌جا باشد: ممکن است پیچیده‌ترین مسئله‌ای که بشر با آن روبه‌رو می‌شود، نه طراحی هوشی برتر، بلکه پاسخ به این پرسش باشد که اگر روزی، دیگر باهوش‌ترین موجود روی زمین نباشیم، می‌خواهیم چه کسی باشیم؟

آینده هنوز نوشته نشده است. اما برای نخستین بار، به اجبار باید بپذیریم که نویسنده آن، تنها ما نخواهیم بود.

و با این حال، چه می‌توان گفت؟ مگر نه این‌که بشر همواره برای غلبه بر ناتوانی‌هایش ابزار ساخته است؟ شاید هوش مصنوعی نیز چیزی جز عصای دست ما نباشد؛ ابزاری برای پیمودن مسیری که به تنهایی از طی کردنش ناتوانیم. اما چه می‌شود اگر همین عصا، روزی چنان قدرتمند شود که دیگر نتوانیم آن را زمین بگذاریم؟ چه می‌شود اگر ابزاری که برای ادامه مسیر ساخته‌ایم، سرانجام راه را بر خودمان ببندد؟

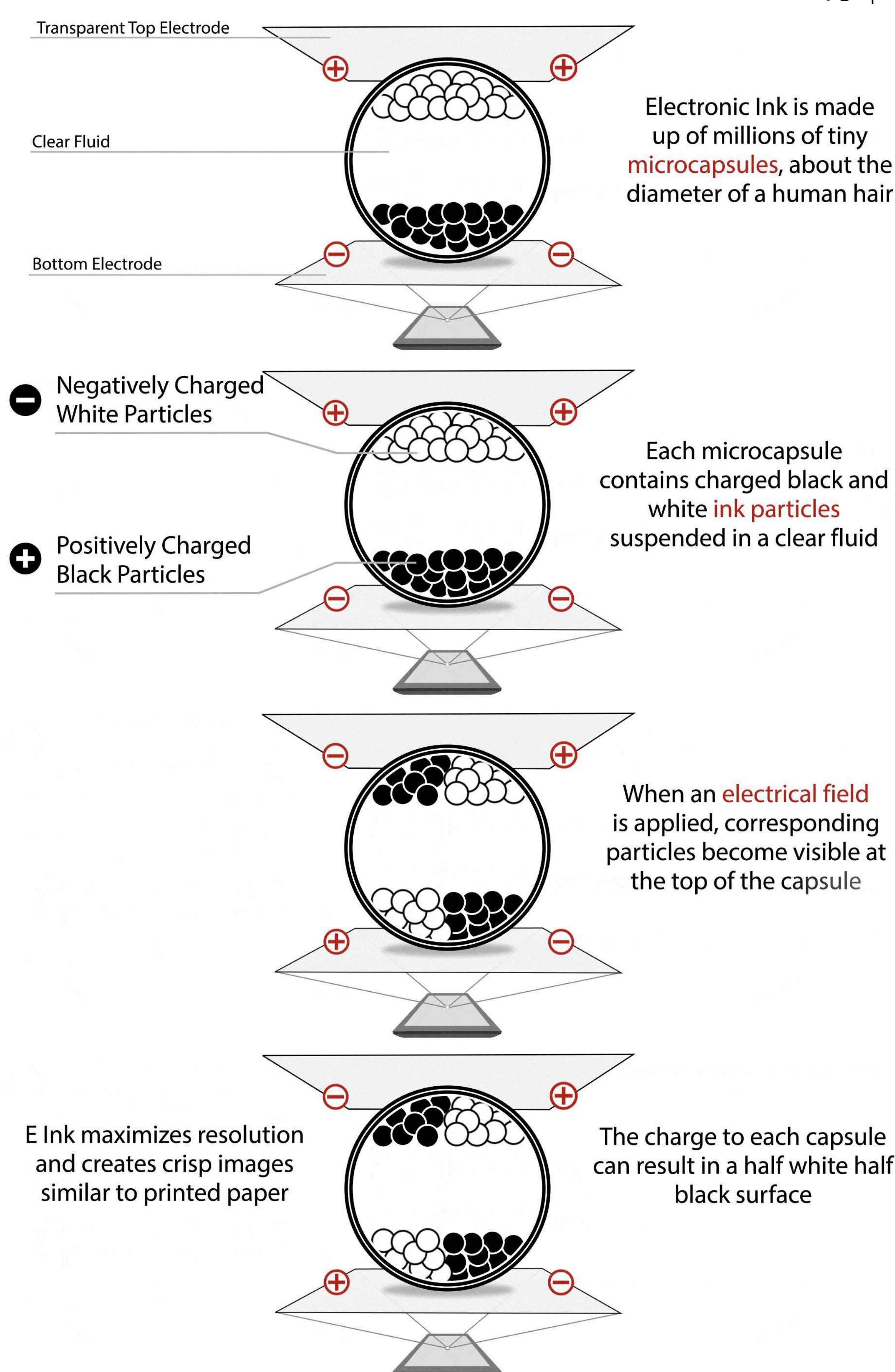
شاید همان چیزی که قرار بود ما را از محدودیت‌هایمان رها کند، روزی به عامل نابودی‌مان تبدیل شود یا به قول بیل گیتس:

«آیا ممکن است یک ماشین تصمیم بگیرد که انسان‌ها تهدیدی هستند، نتیجه بگیرد که منافعش با منافع ما متفاوت است، یا به سادگی دیگر اهمیتی برای ما قائل نباشد؟»

یک تصویر، میلیون‌ها ذره سرگردان

برخلاف نمایشگرهای LCD یا OLED که در آن‌ها هر پیکسل مانند یک منبع نوری کوچک به سمت چشم کاربر نور می‌تاباند، نمایشگر جوهر الکترونیکی مجموعه‌ای میکروسکوپی از ذرات فیزیکی است. در این صفحات، میلیون‌ها کیپسول بسیار ریز (یا حفره‌های میکروکاپ) میان دو لایه الکتروود قرار گرفته‌اند که درون آن‌ها مایعی شفاف و ذرات رنگی باردار شناورند؛ در مدل‌های متداول، ذرات سفید دارای بار مثبت و ذرات سیاه دارای بار منفی هستند.

هنگامی که به الکتروود زیرین ولتاژ اعمال می‌شود، ذرات هم‌نام دفع و ذرات ناهم‌نام جذب می‌شوند؛ در نتیجه، ذرات سفید بالا می‌آیند یا ذرات سیاه جای آن‌ها را می‌گیرند تا حروف شکل بگیرند. آن‌چه چشم انسان می‌بیند، بازتاب نور محیط از روی همین ذرات واقعی است. البته بسیاری از کتاب‌خوان‌ها برای مطالعه در تاریکی به «چراغ جلویی»^۳ مجهز هستند، اما تفاوت کلیدی در زاویه تابش است: این نور نه از پشت صفحه به مردمک چشم، بلکه از لبه‌ها به سطح صفحه می‌تابد و بازتاب ملایم آن به چشم می‌رسد [۱، ۲].



«جریان برق، آرایش ذرات را تغییر می‌دهد؛ اما ذرات، تصویر را بدون نیاز به برق نگه می‌دارند.» [۱]

نمایشگری که کندتر و گران‌تر است؛ پس چرا هنوز می‌خواهیمش؟

دکتر سولماز سلیمی
استاد مدعو
دانشگاه صنعتی شریف
۱۱ دقیقه

فرض کنید در حال مطالعه یک سند محرمانه یا پیش‌نویس مقاله‌ای هستید که هنوز نباید کسی از محتوای آن باخبر شود. ناگهان در اتاق باز می‌شود و فردی وارد می‌شود. شتاب‌زده دکمه خاموش را می‌فشارید؛ دستگاه خاموش می‌شود، اما متن، با خونسردی تمام، همچنان روی صفحه باقی مانده است.

اگر با کتاب‌خوان‌های الکترونیکی کار کرده باشید، این صحنه برایتان آشناست. حتی اگر تاکنون کتاب‌خوان در دست نگرفته باشید، احتمالاً نسخه کوچک‌تر این فناوری را در فروشگاه‌های زنجیره‌ای کشورهای دیگر دیده‌اید: همان برچسب‌های دیجیتال قیمت (ESL)^۱ روی لبه قفسه‌ها که آن‌قدر واضح و بی‌نورند که مخاطب شک می‌کند کاغذ چاپی هستند یا نمایشگر دیجیتال (هرچند جای پرسش است که چرا هنوز در فروشگاه‌های ایران سراغ آن‌ها نرفته‌اند؟).

صفحه جوهر الکترونیکی^۲ پس از قطع کامل جریان برق نیز تصویر را همچون یک برگه چاپ‌شده نگه می‌دارد. همین ویژگی، پرسشی جدی را در ذهن هر دانشجو و مهندس شکل می‌دهد: چه تفاوتی میان ساختار این نمایشگر با صفحه گوشی و لپ‌تاپ وجود دارد که هم چنین رفتار عجیبی از خود نشان می‌دهد، هم کندتر است، هم هنگام تعویض صفحه چشمک می‌زند و هم در کمال تعجب، برای خرید یک نسخه سیاه‌وسفید آن باید هزینه‌ای هم‌سنگ یک تبلت تمام‌رنگی پرداخت کرد؟



ضخامت لایه کپسول‌ها باعث می‌شود کل پنل از رده خارج شود. شرکت E Ink در گزارش‌های سرمایه‌گذاران خود صراحتاً به چالش بازده تولید در گسترش پنل‌های بزرگ‌تر اشاره کرده است [۵].

علاوه بر این، مقیاس بازار نقش تعیین‌کننده‌ای دارد. سالانه صدها میلیون پنل LCD و OLED برای تلفن‌های همراه و تلویزیون‌ها تولید می‌شود که هزینه سرمایه‌گذاری اولیه را سرشکن می‌کند؛ اما بازار کاغذ الکترونیکی بازاری تخصصی با تیراژ محدودتر است که در آن اکثر برندهای شناخته‌شده (از Kindle و Kobo گرفته تا Boox و reMarkable) تولیدکننده لایه نمایشگر نیستند، بلکه خریدار یک زنجیره تأمین متمرکز به‌شمار می‌روند.

وقتی قلم وارد می‌شود؛ چرا نوشتن روی شیشه حس کاغذ را ندارد؟

برای یک دانشجو یا پژوهشگر، کاربری این دستگاه‌ها تنها به روخوانی محدود نمی‌شود؛ حاشیه‌نویسی روی مقالات، حل روابط ریاضی و ترسیم نمودار بخش جدایی‌ناپذیر مطالعه هستند. اما چرا بسیاری از کاربران پس از تهیه تبلت‌های معمولی و قلم‌های دیجیتال، دوباره به دفترهای کاغذی بازمی‌گردند؟

پاسخ در مهندسی سطح و علوم اعصاب نهفته است. حرکت نوک قلم پلاستیکی روی شیشه صیقلی تبلت، اصطکاکی نزدیک به صفر دارد. افزون بر این، ضخامت شیشه محافظ باعث ایجاد فاصله دیداری^۵ میان نوک قلم و خط دیجیتال می‌شود و عضلات دست برای کنترل لغزش قلم، نیازمند انقباض بیش‌تری هستند [۱۲].

پژوهش‌های علوم اعصاب نیز نشان می‌دهند که بازخورد لمسی هنگام نوشتن، با فرایند ثبت اطلاعات در مغز ارتباط دارد. در یک مطالعه تصویربرداری مغزی^۶ توسط اومه‌جیما و همکاران، مشخص شد افرادی که مطالب را روی کاغذ ثبت می‌کنند، نسبت به گروهی که با قلم روی تبلت یا تلفن همراه کار می‌کنند، هم سرعت بالاتری دارند و هم هنگام یادآوری اطلاعات، فعالیت قوی‌تری در ناحیه هیپوکامپ مغز (مرتبط با حافظه و کدگذاری فضایی) نشان می‌دهند [۱۳]. اگرچه در برخی تکالیف آزمایشگاهی ساده‌تر، تفاوت نمره یادگیری میان قلم جوهری و قلم دیجیتال معنادار گزارش نشده است [۱۴]، اما شواهد نشان می‌دهند اصطکاک سطح و ثبات فضایی صفحه تجربه شناختی متفاوتی رقم می‌زند.

معمای قیمت؛ مگر پتنت‌ها در سال‌های ۲۰۱۷ و ۲۰۱۹ منقضی نشدند؟

وقتی یک دفتر دیجیتال یا کتاب‌خوان E Ink کنار یک تبلت معمولی قرار می‌دهیم، نخستین پرسش به برچسب قیمت آن بازمی‌گردد: چرا دستگاهی با پردازنده ساده‌تر و صفحه سیاه‌وسفید، هم قیمت یا حتی گران‌تر از یک تبلت چندمنظوره است؟

در سال‌های ۲۰۱۷ و ۲۰۱۹، موجی از خوش‌بینی میان علاقه‌مندان سخت‌افزار شکل گرفت. علت، آن بود که پتنت پایه «نمایشگر الکتروفوریتیک میکروکپسولی» (ثبت شده توسط پژوهشگران دانشگاه MIT در سال ۱۹۹۷ با شماره US۵۹۶۱۸۰۴A) و یکی از پتنت‌های کلیدی شرکت E Ink (شماره US۶۱۷۲۷۹۸B۱) به پایان دوره اعتبار قانونی خود رسیدند. بسیاری انتظار داشتند با انقضای این اسناد، انحصار بازار بشکند، کارخانه‌های رقیب وارد میدان شوند و قیمت‌ها به شدت کاهش یابند. اما چرا چنین اتفاقی رخ نداد؟

نخست؛ شبکه‌ای از ۱۹۰۰ پتنت فعال

آنچه در سال‌های ۲۰۱۷ و ۲۰۱۹ منقضی شد، تنها دو سند کلیدی بود، نه تمام دارایی فکری این صنعت. طبق گزارش رسمی شرکت E Ink، این مجموعه تا پایان ژوئیه ۲۰۲۶ حدود ۱۹۰۰ پتنت صادرشده و معتبر در آمریکا، تایوان و چین در اختیار داشته است. این مجموعه حقوقی از فرمول شیمیایی سیال و پوشش ضدبازتاب گرفته تا مدارهای راه‌انداز، الگوریتم‌های موج‌فرم و ساختار پنل‌های رنگی را در بر می‌گیرد [۶]. انقضای یک پتنت پایه، به معنای گشوده‌شدن دروازه خط تولید به روی رقبا نیست.

دوم؛ تفاوت فرمول ثبت‌شده با خط تولید

صفحه کتاب‌خوان یک ورق جوهر تنها نیست؛ بلکه ساختاری چندلایه متشکل از فیلم جوهر، صفحه ترانزیستورهای لایه نازک (TFT^۳)، تراشه راه‌انداز، لایه لمسی (معمولاً مبتنی بر فناوری الکترومغناطیسی برای قلم) و لایه هدایت نور است [۴]. چالش اصلی در این جا، نرخ بازده تولید^۴ در ابعاد بزرگ است. به محض آن‌که از تولید برچسب قیمت ۲ اینچی به سمت تولید پنل‌های ۱۰ یا ۱۳ اینچی (مناسب برای خواندن مقالات علمی و فایل‌های PDF) حرکت می‌کنیم، کوچک‌ترین ناهماهنگی در

شاهکار ماجرا این جاست: این فناوری دواپدار^۱ است؛ یعنی برای تکان دادن ذرات به برق نیاز دارید، اما وقتی هر ذره سر جایش نشست، در همان مایع غلیظ معلق می‌ماند و حتی اگر سیم باتری دستگاه را با قیچی ببرید، متن روی صفحه دست‌نخورده باقی می‌ماند! [۲]

همین ویژگی اعجاب‌انگیز است که این روزها سر از اتیکت‌های الکترونیکی قیمت در فروشگاه‌های بزرگ کشورهای دیگر در آورده است (که البته جای پرسش است چرا هنوز در فروشگاه‌های ایران سراغش نرفته‌اند؟). در این هایپرمارکت‌ها، قیمت قفسه‌ها روی نمایشگرهای کوچکی با کنتراست فوق‌العاده بالا ثبت شده است که از نزدیک شبیه کاغذ چاپی به نظر می‌رسند. این‌ها همان پنل‌های جوهر الکترونیکی‌اند؛ نمایشگرهایی که تنها با یک باتری ساعتی کوچک، ۵ تا ۷ سال روشن می‌مانند! مدیر فروشگاه با فشردن یک کلید در نرم‌افزار، قیمت ۱۰ هزار قلم کالا را هم‌زمان عوض می‌کند؛ ذرات تکان می‌خورند، قیمت جدید نقش می‌بندد و تا تخفیف بعدی، هیچ میلی‌آمپر برقی مصرف نمی‌شود.

همین جابه‌جایی فیزیکی، علت کندی و چشمک‌زدن صفحه نیز هست. برخلاف پیکسل‌های کریستال مایع که در کسری از میلی‌ثانیه تغییر وضعیت می‌دهند، در این جا ذرات جامد باید مسافت مشخصی را درون یک سیال غلیظ طی کنند. برای آن‌که ذرات در میانه راه متوقف نشوند و سایه‌های خاکستری ناخواسته نسازند، تراشه کنترل‌کننده هنگام ورق‌زدن صرفاً یک پالس ساده ارسال نمی‌کند؛ بلکه دنباله‌ای محاسبه‌شده از ولتاژهای متناوب، موسوم به موج‌فرم به‌روزرسانی را اعمال می‌کند. آن چشمک سیاه‌وسفید هنگام ورق‌زدن، ناشی از درگیر شدن پردازنده نیست؛ بلکه فرایند بازنشانی آرایش ذرات برای جلوگیری از باقی ماندن رد تصویر قبلی یا همان تصویر شبح‌وار^۲ است [۱۱، ۲].

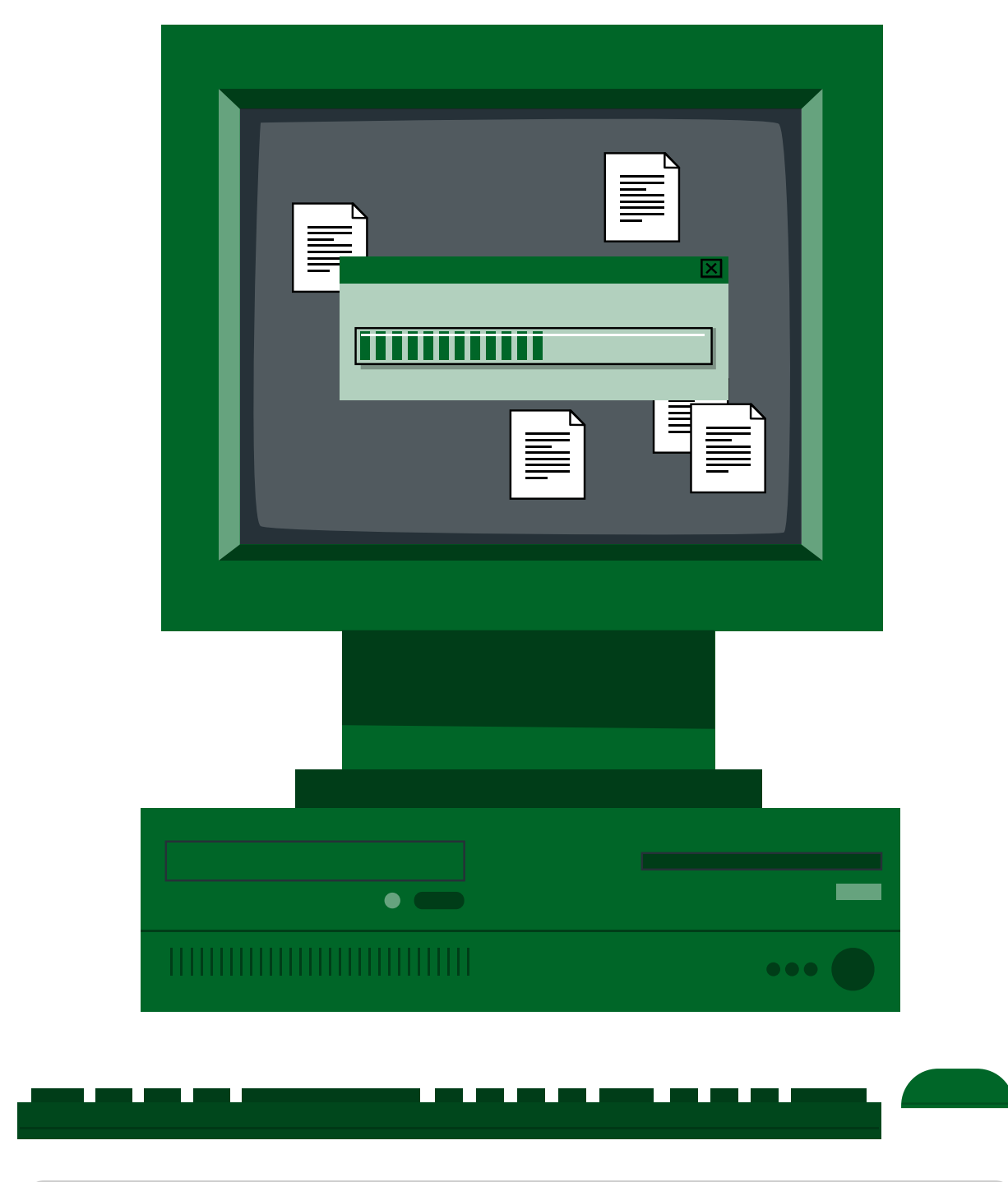
البته سرعت این فناوری ثابت نمانده است؛ برای نمونه، در نسل‌های رنگی جدیدتر مانند Gallery 3 که از چهار نوع ذره رنگی استفاده می‌کنند، زمان به‌روزرسانی سیاه‌وسفید به حدود ۳۵۰ میلی‌ثانیه رسیده است. با این حال، مصالحه مهندسی همچنان پابرجاست؛ نمایشگری که برای متن ایستا کم‌نظیر است، گزینه مناسبی برای پخش ویدیو و تصاویر متحرک نخواهد بود [۳].

منابع

- [۱] E Ink Corporation, How E Ink Works
- [۲] E Ink Corporation, Bistability & Power Benefits
- [۳] E Ink Corporation, E Ink Gallery Launch
- [۴] E Ink ESG & Supply Chain Report, Display Module Layers (۲۰۲۴)
- [۵] E Ink Investor Relations, Large-Panel Yield Challenges
- [۶] E Ink Investor Relations, IP Management; patents US۵۹۶۱۸۰۴A, US۶۱۷۲۷۹۸B۱
- [۷] Ben-Yehudah & Brann (۲۰۱۹). Digital Text Comprehension in Students with ADHD. Res. Dev. Disabil.
- [۸] Stern & Shalev (۲۰۱۳). Display Medium and Reading in Adolescents with ADHD. Res. Dev. Disabil.
- [۹] Benedetto et al. (۲۰۱۳). E-readers and Visual Fatigue. PLOS ONE
- [۱۰] Siegenthaler et al. (۲۰۱۲). LCD vs e-Ink: Fatigue and Visual Strain. Ophthalmic Physiol. Opt.
- [۱۱] Waveshare, E-Paper Manual: Partial Refresh & Ghosting
- [۱۲] Annett et al. (۲۰۱۴). The Pen Is Mightier (Graphics Interface) & E Ink Digital Paper Specs
- [۱۳] Umejima et al. (۲۰۲۱). Paper Notebooks vs. Mobile Devices in Memory Retrieval. Front. Behav. Neurosci.
- [۱۴] Osugi et al. (۲۰۱۹). Digital Pen vs. Ink Pen: An EEG Study. Front. Hum. Neurosci.

۲. **بار شناختی مقاومت در برابر اعلان‌ها:** روی دستگاهی که یک کلیک با شبکه‌های اجتماعی و پیام‌رسان‌ها فاصله دارد، بخشی از ظرفیت اجرایی مغز صرف مقاومت در برابر وسوسه خروج از متن می‌شود.

نمایشگر جوهر الکترونیکی درمان پزشکی برای اختلال توجه نیست؛ اما با تبدیل محدودیت سخت‌افزاری به یک سپر محیطی، به کمک تمرکز می‌آید. در کتاب‌خوان، متن نه با پیمایش عمودی، بلکه صفحه‌به‌صفحه ورق می‌خورد و ثبات فضایی حفظ می‌شود. از سوی دیگر، صفحه سیاه‌وسفید با نرخ نوسازی کند، مرور شبکه‌های اجتماعی و تماشای ویدیو را چنان بی‌جاذبه می‌کند که چرخه پاداش فوری این برنامه‌ها در عمل از کار می‌افتد. در واقع، خریدار این دستگاه بخش زیادی از هزینه خود را می‌پردازد تا امکان حواس‌پرتی را از سخت‌افزار خود حذف کند.



کاغذ الکترونیکی مجموعه‌ای از مصالحه‌های مهندسی است: مصرف انرژی ناچیز برای نگه‌داشتن تصویر در برابر زمان بیش‌تر برای تغییر آن؛ و آرامش چشم و تمرکز عمیق در برابر چشم‌پوشی از رنگ‌های درخشان و انیمیشن‌های سریع. این فناوری قرار نیست جایگزین تلفن همراه یا لپ‌تاپ شود؛ بلکه در میان انبوه نمایشگرهایی که ثانیه‌به‌ثانیه برای ربودن توجه ما نور می‌تابانند، آرام در کناری می‌ایستد و می‌پرسد: اگر قرار باشد یک صفحه هیچ تلاشی برای فریفتن مغز نکند، دوست دارید روی آن چه چیزی باقی بماند؟

مزیت رقابتی تبلت‌های یادداشت‌برداری E Ink دقیقاً در همین نقطه تعریف می‌شود: با حذف شیشه ضخیم و استفاده از پوشش‌های بافت‌دار روی صفحه در کنار نوک قلم‌های نمدی، مقاومت مکانیکی کاغذ را زیر دست کاربر شبیه‌سازی می‌کنند و هم‌زمان امکان ذخیره‌سازی و جست‌وجوی دیجیتال را فراهم می‌آورند [۱۲].

خریدن یک «محدودیت»؛ از خستگی چشم تا تمرکز در افراد دارای ADHD

مهم‌ترین پرسش برای دانشجویی که روزانه ساعت‌ها با متون تخصصی سروکار دارد این است: آیا این نمایشگر واقعاً تمرکز را بهبود می‌بخشد؟

بخش نخست پاسخ به فیزیولوژی بینایی مربوط است. مطالعات مقایسه‌ای میان نمایشگر LCD، صفحه E Ink و کتاب کاغذی نشان می‌دهند که اگرچه از نظر سرعت خواندن، یک پنل E Ink باکیفیت عملکردی مشابه LCD دارد [۱۰]، اما در معیار خستگی دیداری^۱ تفاوت آشکاری دیده می‌شود. مطالعه طولانی‌مدت روی LCD به دلیل تابش مستقیم نور پس‌زمینه، نرخ پلک زدن را کاهش داده و خستگی چشمی بیش‌تری نسبت به کاغذ و E Ink ایجاد می‌کند، در حالی‌که رفتار نوری E Ink بسیار نزدیک به کاغذ چاپی است [۹].

بخش دوم پاسخ، به معماری حواس‌پرتی و چالش نقص توجه و بیش‌فعالی (ADHD) بازمی‌گردد. پژوهش‌های شناختی روی دانشجویان و نوجوانان دارای ADHD نشان داده‌اند که مطالعه متون روی صفحات دیجیتال متداول، در مقایسه با کاغذ، می‌تواند با افت درک مطلب، دشواری در حفظ توجه پایدار^۲ و خطا در مدیریت زمان همراه باشد [۷، ۸].

محیط دیجیتال معمولی از دو مسیر، تمرکز را تهدید می‌کند:

۱. **پیمایش عمودی^۳ در برابر ورق‌زدن:** مغز انسان هنگام مطالعه، یک «نقشه فضایی» از موقعیت جملات در چهارچوب ثابتی از صفحه می‌سازد. پیمایش پیوسته روی تبلت و رایانه، این لنگرگاه تصویری را برهم می‌زند و بار اضافی بر حافظه فعال تحمیل می‌کند.

چگونه میلیون‌ها سوکت TCP را

بدون سرخ کردن CPU هندل کنیم؟

راز هندل کردن میلیون‌ها اتصال هم‌زمان



اگر تا به حال از API های سطح پایین شبکه در زبان C استفاده کرده باشید، حتماً با مجموعهٔ توابع استاندارد از socket، bind و listen گرفته تا conn، ect، accept، send، recv و... مواجه شده‌اید. این توابع بخشی از رابط Berkeley sockets^۱ هستند که در اکثر سیستم‌عامل‌ها، به‌عنوان روش اصلی برای ساختن و استفاده از سوکت‌های شبکه استفاده می‌شود. حالا فرض کنید قصد پیاده‌سازی یک سرور را روی سیستم‌عامل گنو/لینوکس یا یکی از سیستم‌عامل‌های خانوادهٔ BSD^۲ داریم. مستقل از هر زبان برنامه‌نویسی و abstraction ای که استفاده کنیم، در نهایت تمام عملیات مربوط به شبکه از طریق system call های کم‌وبیش متناظر با Berkeley sockets به کرنل داده می‌شوند. پس بیاید برای بررسی بهتر، آن برنامه را به‌صورت شبه‌کدی نزدیک به زبان C بنویسیم. (برای درک بهتر، پارامترهای توابع ساده‌سازی شده‌اند و از برخی خطاها و حالات خاص ممکن چشم‌پوشی شده است):

```
char buffer[4096];
int sockfd = socket(...);
bind(sockfd, "0.0.0.0:8080");
listen(sockfd, ...);

while (true) {
    int connfd = accept(sockfd, ...);

    // Handle the connection
    ssize_t n = recv(connfd, buffer, ...);
    if (n > 0) {
        send(connfd, buffer, n);
    }

    // Close the connection
    close(connfd);
}
```

این برنامه، پیاده‌سازی یک سرور ساده است که صرفاً بخشی از اول پیامی که از طرف هر کلاینت ارسال می‌شود را به آن بر می‌گرداند و سپس اتصال را می‌بندد. سه دستور socket، bind و listen، سوکت سمت سرور را روی پورت ۸۰۸۰ برای گوش دادن به اتصال‌های جدید TCP آماده می‌کنند. سپس در یک حلقهٔ while، با دستور

accept یکی از اتصال‌های جدید را قبول می‌کنیم و سپس تا حداکثر ۴ کیلوبایت داده را از آن می‌خوانیم (دستور recv) و سعی می‌کنیم همان را به کلاینت برگردانیم (دستور send). در نگاه اول شاید به نظر برسد که این کد می‌تواند به چندین کلاینت به‌صورت هم‌زمان پاسخ‌گو باشد، اما اصلاً این‌طور نیست! دستوراتی مانند accept، recv و send ممکن است در صورت آماده نبودن برای انجام عملیات، روند اجرای برنامه را به‌صورت موقت متوقف کنند. مثلاً اگر هنوز اتصالی از طرف هیچ کلاینتی برقرار نشده باشد، دستور accept صبر می‌کند تا اولین اتصال برقرار شود. یا اگر هنوز داده‌ای ارسال نشده باشد، دستور recv صبر می‌کند تا قطعات داده از سمت کلاینت ارسال شوند (مگر این‌که اتصال بسته شود یا خطایی رخ دهد). این اتفاق باعث می‌شود که فرضاً اگر یک کلاینت، اتصالی را باز کند ولی برای چند ثانیه داده‌ای ارسال نکند، برنامه ما در اجرای دستور recv گیر کند و قابلیت قبول کردن و رسیدگی به اتصال‌های جدید را نداشته باشد.

فرض کنید این‌گونه مشکلات در برنامه‌های پیچیده‌تر مانند سرورهای HTTP که باید قابلیت خدمت‌رسانی به هزاران یا حتی میلیون‌ها اتصال هم‌زمان را داشته باشند، چقدر ناگوار خواهد بود. پس به‌راستی آن‌ها چگونه این مشکل را حل می‌کنند؟!

یک راه‌حل ساده، ایجاد یک thread به‌ازای هر اتصال است. یعنی بعد از خروج دستور accept، هر دفعه یک thread جدید درست کنیم تا فقط به همان یک اتصال رسیدگی کند و از نگه‌داشتن بقیهٔ روند برنامه جلوگیری کند. این روش از لحاظ مفهومی بسیار ساده است و برای برنامه‌های آموزشی با نیازمندی‌های پایین کارآمد است. اما مشکل از جایی آغاز می‌شود که تعداد درخواست‌ها بالا رفته و نیاز به ساختن تعداد زیادی thread باشد. از آن‌جایی که این thread ها توسط سیستم‌عامل مدیریت می‌شوند و نیاز به حجم قابل‌توجهی از منابع سیستمی برای مدیریت‌شان دارند؛ تعداد بسیار زیاد آن‌ها باعث افزایش مصرف حافظهٔ اصلی می‌شود و همچنین تعداد context switch ها را بسیار بالا می‌برد. از این رو، باید به‌دنبال راهی باشیم تا با استفاده از تعداد محدودی thread، بتوانیم برای چند سوکت هم‌زمان صبر کنیم.

قبل از رفتن سراغ اصل مطلب، می‌خواهم به یک راه‌حل ساده لوحانه اشاره کنم. سوکت‌ها این قابلیت را دارند که در حالت non-blocking قرار بگیرند. یعنی هیچ‌کدام از دستورات اجرایی روی آن‌ها، منتظر I/O نمی‌مانند و در صورتی که در آن لحظه قابل‌انجام نباشند، خطای EWOULDBLOCK را باز می‌گردانند که یعنی اجرای این دستور نیازمند توقف برنامه و صبر کردن است و برنامه باید بعد آماده شدن سوکت برای اجرای عملیات، دوباره آن دستور را فراخوانی کند. یک راه‌حل ممکن برای مشکل مطرح‌شده، این است که تعداد زیادی اتصال را بپذیریم (با accept) و سپس آن‌ها را در حالت non-blocking قرار دهیم و پشت‌سر هم در یک حلقه، تلاش برای خواندن از آن‌ها کنیم؛ سپس اگر خطای EWOULDBLOCK برای هر کدام از آن‌ها برگردانده شد، از آن سوکت در آن مرتبهٔ اجرای حلقه صرف‌نظر کنیم. این راه‌حل، به‌دلیل اشغال همیشگی پردازنده و سیستم‌عامل در یک حلقهٔ باطل (الگوی polling^۳) معمولاً کاربرد عمومی ندارد. استفاده از توقف‌های کوچک بین هر دور حلقه نیز به‌دلیل ایجاد تأخیر بی‌مورد در پردازش درخواست‌ها، راهکار مطلوبی به‌شمار نمی‌آید.

منجی ما: select

ریشهٔ اصلی مشکل، آن‌جایی است که هر کدام از دستوراتی که تا الان بررسی کردیم، صرفاً روی یک عملیات خاص و روی یک سوکت خاص عمل می‌کنند. در این حالت، وقتی تعداد سوکت‌ها زیاد می‌شود، یا باید بین آن‌ها همواره چرخ باطل بزنیم تا ببینیم کدام سوکت‌ها آمادهٔ اجرای عملیات (مثلاً پذیرش اتصال، خواندن و یا نوشتن) هستند و چرخه‌های پردازنده را هدر بدهیم یا این‌که احتمال از دست دادن یا تأخیر زیاد در اجرای عملیات‌ها روی سوکت‌های دیگر را بپذیریم.

از آن‌جایی که سیستم‌عامل همواره در مورد حالت سوکت‌های مختلف و آمادگی آن‌ها برای عملیات‌های مختلف اطلاع دارد، از لحاظ نظری می‌تواند به ما این امکان را بدهد که هم‌زمان -مثلاً برای خواندن از چند سوکت مختلف- صبر کنیم و هر کدام که آمادهٔ خواندن شدند، به برنامه ما اطلاع بدهد تا کار خود را ادامه دهد. خوشبختانه توسعه‌دهندگان BSD، به فکر این مسئله بودند و یک دستور

برنامه خودمان نیز باید تمام سوکت‌ها را پشت‌سر هم بررسی کنیم که آیا در مجموعه سوکت‌های آماده وجود دارند یا نه. این کار پیچیدگی زمانی $O(n)$ دارد که n تعداد کل سوکت‌هاست، نه فقط سوکت‌های آماده. علاوه بر این دلایل، چون خروجی این دستور مجموعه‌های ورودی را تغییر می‌داد، پیش از هر بار فراخوانی این دستور نیاز به ساختن مجدد ساختار `fd_set` بود.

به دلیل این محدودیت‌ها، مدتی بعد، از دستور جدیدی رونمایی شد که برخی از این مشکلات را حل می‌کرد.

poll

در این دستور جدید که `poll` نام داشت، به جای ورودی دادن مجموعه به صورت یک `bitset`، لیست از `pollfd` ها ارسال می‌شود که تقریباً به این صورت تعریف شده است:

```
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

به این صورت که شماره `file descriptor` در `fd`، نوع عملیاتی که منتظر آمادگی آن هستیم (خواندن، نوشتن یا ...) در `events` و خروجی دستور در `revents` قرار خواهند گرفت. از جهات دیگر، فراخوانی این دستور شبیه `select` است.

این ساختار جدید، برخی از مشکلاتی که هنگام استفاده از دستور `select` مطرح می‌شوند را حل کرد. در نهایت نیز هر دو دستور به استاندارد POSIX راه یافتند و واقعی به صورت `system call` قابل استفاده هستند.

با این حال، حتی `poll` نیز بعد از چند هزار اتصال شروع به نشان دادن ضعف‌هایش^۲ می‌کند. مشکل از آنجایی ناشی می‌شود که بین هر بار فراخوانی این `system call`، سیستم عامل نمی‌داند که ورودی آن تغییر کرده است یا نه. یعنی هر دفعه، کرنل باید چندین هزار ورودی آن را بررسی کند، زیرا این آرایه در فضای کاربر قرار دارد و ممکن است بین فراخوانی‌های مختلف، بدون اطلاع مستقیم کرنل تغییر کند. در واقع همان مشکل بررسی از $O(n)$ هنوز در سطح کرنل وجود دارد. در ادامه به بررسی دواي این درد می‌پردازیم!

آن مجموعه سوکت‌ها که به آن‌ها اشاره شد، با تایپ کردن `fd_set` نشان داده می‌شوند. ما یک مجموعه از تمام سوکت‌هایی که تمایل به خواندن از آن‌ها داریم را در `connections` نگه می‌داریم. سپس در هر دور حلقه، یک کپی از آن مجموعه برای پاس دادن به دستور `select` ایجاد می‌کنیم. دلیل این کار این است که این دستور، مجموعه‌های ورودی خودش را تغییر می‌دهد و فقط سوکت‌هایی که آماده هستند را در آن‌ها نگه می‌دارد و بقیه را حذف می‌کند.

از آنجایی که حالت «قابل خواندن» روی یک سوکت در حال گوش دادن به معنی آماده بودن یک اتصال جدید است، می‌توانیم از همین سازوکار برای `accept` کردن بدون درنگ اضافه نیز استفاده کنیم. به این صورت که اگر سوکت اولیه در مجموعه سوکت‌های آماده خواندن بود، دستور `accept` را اجرا می‌کنیم؛ سپس تمام سوکت‌های اتصال را بررسی می‌کنیم و اگر قابل خواندن بودند، از آن‌ها می‌خوانیم و ادامه کار لازم را انجام می‌دهیم.

لازم به ذکر است که در یک برنامه واقعی، لازم است امکان درنگ در عملیات نوشتن (`send`) نیز در نظر گرفته شود. همچنین، حتی بعد از اطلاع از آمادگی یک عملیات بعد از `select`، در شرایط خاصی ممکن است باز هم نیاز به توقف برنامه باشد و در سناریوهای جدی‌تر باید این موارد نیز لحاظ شوند، اما این‌جا برای سادگی فرض می‌کنیم این اتفاقات رخ نمی‌دهند.

با استفاده از دستور `select`، می‌توان از مرتبه صدها اتصال را به راحتی مدیریت کرد، اما با افزایش بیش‌تر تعداد اتصالات، محدودیت‌های آن آشکار می‌شود. بزرگ‌ترین ضعف این دستور، در نحوه نمایش مجموعه سوکت‌ها در ساختار `fd_set` است. در پیاده‌سازی لینوکس، این ساختار مجموعه سوکت‌ها را در یک `bitset` نگه‌داری می‌کند. از آنجایی که هر سوکت یک `file descriptor` خاص خود را دارد - که یک عدد صحیح است - می‌توان با تنظیم بیت مربوط به هر سوکت، آن را در مجموعه انتخاب کرد. به دلیل این که تعداد بیت‌های موجود در این `bitset` محدود است (معمولاً ۱۰۲۴ بیت)، نمی‌توان `file descriptor` های با مقدار عددی بیش‌تر از ۱۰۲۳ را انتخاب کرد. از طرف دیگر، در

دیگر به نام `select` را طراحی کردند. با کمی ساده‌سازی می‌توان در نظر گرفت که با این دستور، برنامه شما به سیستم عامل می‌گوید:

«من دوست دارم روی این مجموعه از سوکت‌ها عملیات خواندن و روی این یکی مجموعه، عملیات نوشتن را انجام دهم. هر وقت حداقل یکی از آن‌ها آماده آن کار بودند، به من اطلاع بده! و در ضمن، بیش‌تر از `tt` ثانیه هم طولش نده!»

این دستور (با کمی ساده‌سازی)، دو مجموعه از سوکت‌ها و یک مقدار `timeout` را ورودی می‌گیرد. یکی برای خواندن و دیگری برای نوشتن و فقط وقتی تمام می‌شود که یا حداقل یکی از عملیات‌های خواسته شده قابل انجام شوند یا `timeout` خواسته شده گذشته باشد.

برای مثال، می‌توانیم شبه‌کد بالا را طوری ارتقاء دهیم که برای خواندن از اتصال‌ها، جداگانه صبر نکند و به محض آمادگی هر کدام‌شان، به سرعت از آن بخواند و ادامه کار را انجام دهد.

```
char buffer[4096];
int sockfd = socket(...);
int max_fd = sockfd;

bind(sockfd, "0.0.0.0:8080");
listen(sockfd, ...);

fd_set connections;
FD_ZERO(&connections);
FD_SET(sockfd, &connections);

while (true) {
    fd_set ready = connections;
    select(max_fd + 1, &ready, NULL, NULL, NULL);

    // Check for new connections
    if (FD_ISSET(sockfd, &ready)) {
        int connfd = accept(sockfd, ...);
        FD_SET(connfd, &connections);
        if (connfd > max_fd) {
            max_fd = connfd;
        }
    }

    // Handle existing connections
    for (int fd = 0; fd <= max_fd; fd++) {
        if (fd == sockfd || !FD_ISSET(fd, &ready))
            continue;

        ssize_t n = recv(fd,
            buffer, sizeof(buffer), ...);

        if (n <= 0) {
            // Close the connection
            close(fd);
            FD_CLR(fd, &connections);
            continue;
        }

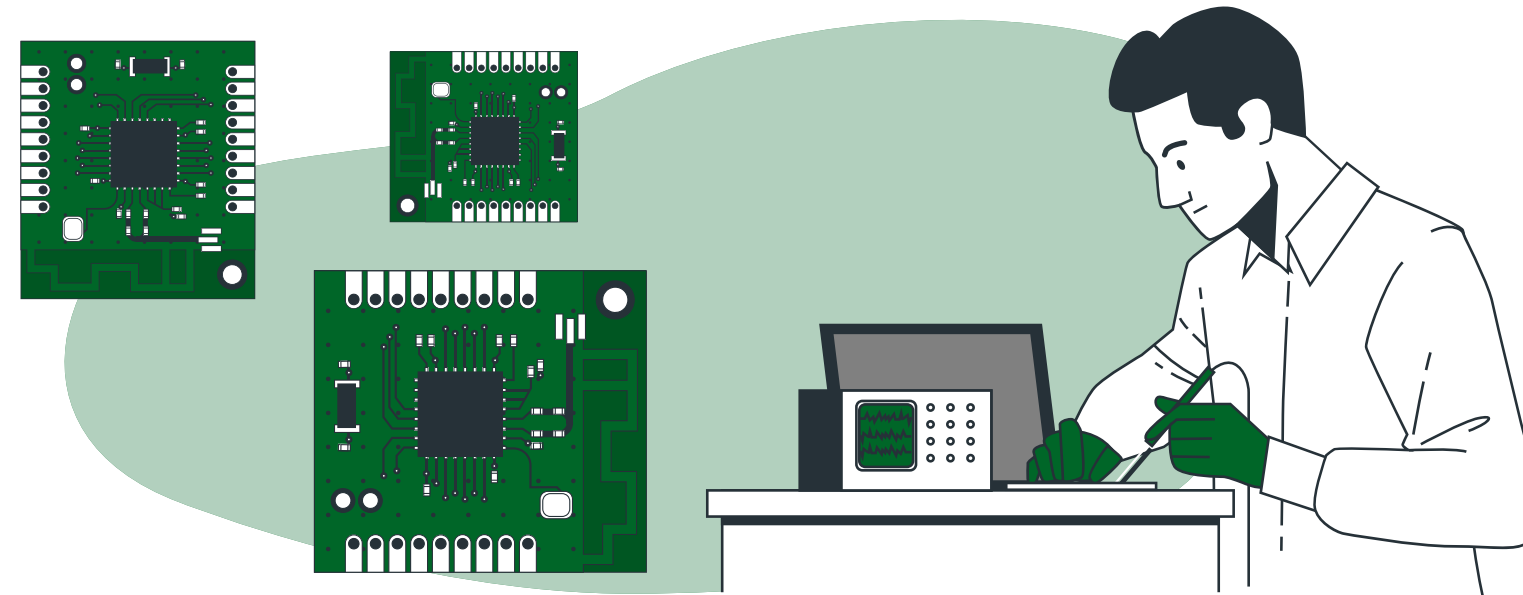
        send(fd, buffer, n, ...);
    }
}
```


اعمال تغییر در ورودی‌شان بروز می‌دادند، این دستور خروجی خود را در آرایهٔ events ذخیره می‌کند. همچنین مانند دو دستور POSIX، پارامتر timeout نیز به چشم می‌خورد.

علاوه بر این سه دستور، چند دستور دیگر مربوط به epoll نیز وجود دارند که اغلب نسخه‌های متفاوتی از همین سه دستور هستند. برای خواندن بیشتر در مورد آن‌ها، می‌توانید به manpage مربوط به (۷) epoll^۳ مراجعه کنید!

در نهایت، این API ها به استاندارد جدید نرم‌افزارهای سمت سرور تبدیل شدند و در کنار بسیاری از فناوری‌های دیگر، اجازهٔ رسیدگی به میلیون‌ها اتصال TCP از روی یک سرور را می‌دهند. برنامه‌ها، کتاب‌خانه‌ها و runtime‌هایی مانند:

- nginx^۴
- Apache HTTP Server^۵
- Redis^۶
- Node.js^۷ / libuv^۸
- Tokio^۹ / Mio^{۱۰}
- Python's asyncio^{۱۱}



تنها بخش کوچکی از نرم‌افزارهایی هستند که از رابط epoll و kqueue و مشابه آن‌ها استفاده می‌کنند. نکتهٔ جالب این‌جاست که در بسیاری از runtime های مدرن async I/O، می‌توان از مفهومی مشابه thread استفاده کرد؛ در حالی‌که در زیر پوست برنامه، async I/O runtime کار سخت را برای ما انجام می‌دهد و برنامهٔ ما را حتی روی یک thread واقعی، قادر به رسیدگی به تعداد بسیار زیادی از درخواست‌ها می‌کند. به چنین تسک‌های سبک‌وزن و زمان‌بندی‌شده‌ای در فضای کاربر، بسته به runtime و مدل اجرایی، گاهی اصطلاحاتی مانند coroutine یا green thread نیز اطلاق می‌شود. برای مثال در زبان Rust و با Tokio runtime می‌توان به این صورت، این برنامه را نوشت (با این تفاوت که کل بایت‌های ارسال شده بازتاب می‌شوند):

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

# [tokio::main(flavor = "current_thread")]
async fn main() → std::io::Result<()> {
    let listener = TcpListener::bind("0.0.0.0:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buffer = [0; 4096];

            loop {
                let bytes_read = socket.read(&mut buffer).await.unwrap();
                if bytes_read == 0 {
                    break;
                }

                socket.write_all(&buffer[..bytes_read]).await.unwrap();
            }
        });
    }
}
```

همان‌طور که می‌بینید، ساختار این برنامه مانند اولین راه‌حل ساده‌انگارانه است که از thread های اضافه استفاده می‌کرد. همچنین این‌جا، هنگام فراخوانی دستوراتی که ممکن است منتظر بمانند، از کلمهٔ کلیدی await استفاده شده است. این ساختار به ما اجازه می‌دهد که تظاهر کنیم از همان الگوی سادهٔ thread استفاده می‌کنیم، در حالی‌که اتفاقی که واقعاً می‌افتد، کمی پیچیده‌تر و بسیار بهینه است.

epoll، kqueue و...

ایدهٔ اصلی رفع این مشکل بسیار ساده است: ساختار دادهٔ مجموعهٔ سوکت‌ها را به کرنل منتقل کنید و تمام عملیات‌هایی که آن را تغییر می‌دهند را به صورت system call در بیاورید. با این روش، کرنل همواره از محتویات آن مجموعه اطلاع دارد و می‌تواند به جای بررسی تک‌تک آن‌ها هنگام فراخوانی system call، به‌طور مستقیم وقتی فعالیت I/O روی سوکت‌های مشخص شده رخ می‌دهد، تکلیف ما را مشخص کند.

کرنل‌های لینوکس و سیستم‌عامل‌های خانوادهٔ BSD هر دو این الگورا در قالب API های خاص خود پیاده‌سازی کرده‌اند. با این‌که ظاهر این API ها با هم تفاوت قابل‌توجهی دارد، ایدهٔ اصلی پشت‌شان یکسان است. راه‌حل‌های لینوکس و BSD به ترتیب epoll^۱ و kqueue^۲ نام دارند.

در ادامه به بررسی بخش‌های اصلی رابط epoll که به عنوان چند system call لینوکس در دسترس است، می‌پردازیم. از اولین دستور این رابط، برای ساختن ساختار دادهٔ سمت کرنل استفاده می‌شود:

```
int epoll_create1(int flags);
```

این دستور یک file descriptor بر می‌گرداند که برای اشاره به آن ساختار داده استفاده می‌شود.

دومین دستور، به ما اجازهٔ ایجاد تغییرات در آن مجموعه را می‌دهد:

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
```

با هر بار فراخوانی این دستور، می‌توان یک سوکت (fd) را به آن مجموعه اضافه یا از آن کم کرد یا در پارامترهای آن تغییراتی ایجاد کرد. ساختار epoll_event دارای یک فیلد events است که اجازهٔ مشخص کردن نوع عملیاتی که مایل به انجام آن هستیم را به ما می‌دهد (مانند ساختار pollfd که بالاتر به آن اشاره شد).

می‌توان گفت دستور سوم، اصلی‌ترین دستور این رابط است که اصل کار صبر کردن را انجام می‌دهد.

```
int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int timeout);
```

برخلاف poll و select که خروجی خود را با

مهندسی نرم افزار در عصر هوش مصنوعی



تصور کنید ده سال پیش است و شما در یک شرکت نرم افزاری مشغول به فعالیت هستید. یک باگ باعث می شود شب ها بی خوابی بکشید و ماه ها، صدها هزار خط کد را بارها مرور کنید. شاید در آن زمان اگر کسی به شما می گفت که روزی قرار است ابزاری داشته باشید تا نشست های طولانی دیباگ شما را به فرایندی چندساعته تبدیل کند، باور نمی کردید. اما امروز، این واقعیتی است که آن را زندگی می کنیم. سال گذشته، مقاله ای^۱ در یکی از خبرگزاری های برنامه نویسان نوشته شد که یک توسعه دهنده ارشد C++ در یکی از شرکت های FAANG (سرنام پنج شرکت بزرگ فناوری آمریکا: Facebook (متا)، Amazon، Apple، Netflix و Google)، یک باگ چهارساله که بیش از ۲۰۰ ساعت روی دیباگ آن تلاش شده بود را به کمک مدل Claude Opus 4 تنها در چند ساعت با کم تر از ۳۰ پیام حل کرد.

آن چه بدیهی است، این است که شکل مشاغل چون مهندسی نرم افزار در حال تغییر است. صرف کد و کدنویسی دیگر ارزش سابق را ندارد، چرا که ابزارهای هوش مصنوعی می توانند آن را انجام دهند و حتی در مواردی بهتر از انسان ها عمل کنند. سال پیش بود که طی صحبت با تعدادی از متخصصان، اساتید و دانشجویان، مثالی جالب و تکرارشونده می شنیدم. زمانی که کامپیوترها به ادارات آمدند، عده ای در برابر یادگیری کار کردن با آن ها به شدت مقاومت می کردند و خود را با شرایط تطبیق نمی دادند. امروزه اما در ادارات، کم تر کسی را می بینید که کار کردن با کامپیوتر و گوشی های هوشمند را -حداقل در حد پیش بردن فرایندهای اداری و کارهای روزمره- بلد نباشد. از این مثال، یک نتیجه واضح می توان گرفت که ما را به یاد نظریه گزینش طبیعی داروین می اندازد؛ «بقای بهترین ها» یا به عبارتی «در طول نسل ها، جمعیت با محیط خود سازگار می شود» و این یعنی آدم های ناسازگار با شرایط (یا تغییر آن ها) به طور خودکار از محیط حذف می شوند.

در این مقاله، با هم ۱۰ مهارتی را مرور می کنیم که می توانند به طرز چشمگیری در سازگاری شما با شکل جدید مهندسی نرم افزار در عصر هوش مصنوعی مؤثر باشند.

۱. مهارت های نرم

با افزایش توانمندی ابزارهای هوش مصنوعی در کارهای فنی، اهمیت مهارت های انسانی غیرقابل جایگزین، بیش از پیش برجسته می شود. مهندسی نرم افزار منحصر به کدنویسی نیست؛ بلکه استخراج نیازهای مبهم ذی نفعان، تبادل نظر فنی، ساده سازی مفاهیم پیچیده و ایجاد هم گرایی میان تیم ها را نیز در بر می گیرد.

اگرچه هوش مصنوعی در تولید مکاتبات، مستندسازی و تدوین دستور جلسات مؤثر است، اما تعیین هدف، مخاطب و چرایی ارتباطات همچنان بر عهده انسان است.

از این رو، ارتباط مؤثر، کار تیمی، مذاکره، ارائه، مدیریت تعارض و پرسش گری دقیق از ارکان اصلی مهارت های مهندس نرم افزار محسوب می شوند. با کاهش هزینه تولید کد، توانایی درک انسان ها و انتقال مفاهیم ارزشی دوچندان می یابد.

۲. مهارت های T شکل

یکی از خطرهایی که ابزارهای هوش مصنوعی ایجاد کرده اند، ایجاد توهم دانش است. ممکن است بدون این که واقعاً Docker را بشناسید، از یک مدل بخواهید برایتان Dockerfile بنویسد. بدون شناخت Kubernetes یک Deployment ایجاد کنید و بدون اینکه بدانید Index در پایگاه داده دقیقاً چه کاری انجام می دهد، از هوش مصنوعی بخواهید پرسمان شما را بهینه کند. احتمالاً در بسیاری از مواقع نیز خروجی کار می کند. مشکل از جایی شروع می شود که کار نمی کند.

همین جاست که داشتن مهارت های T شکل اهمیت پیدا می کند. یک مهندس T شکل، در یک یا چند حوزه دانش عمیقی دارد، اما در کنار آن از حوزه های اطراف نیز شناخت کافی دارد. برای مثال ممکن است تخصص اصلی شما توسعه بک اند باشد، اما از شبکه، سیستم عامل، پایگاه داده، امنیت، DevOps و فرانت اند نیز آن قدر بدانید که بتوانید مسئله را در سطح یک سیستم ببینید. در گذشته شاید می توانستید بگویید «این قسمت کار من نیست». اما زمانی که

یک دستیار هوش مصنوعی می تواند ظرف چند دقیقه در بک اند، فرانت اند، پایگاه داده و زیرساخت تغییر ایجاد کند، کسی باید بتواند ارتباط میان تمام این تغییرات را بفهمد.

۳. قدرت حل مسئله و نوآوری

در صورتی که مسئله کاملاً شفاف و دارای محدودیت های مشخص باشد، هوش مصنوعی عملکرد مطلوبی دارد؛ اما چالش های واقعی صنعت نرم افزار غالباً مبهم و فاقد تعریف دقیق هستند.

ارزش مهندس نرم افزار در تفکیک عوارض ظاهری (نظیر کندی سامانه) از علل ریشه ای (مانند ساختار نامناسب API، پرسمان های بهینه نشده یا فرایند نادرست کسب و کار) نهفته است.

توانایی شکستن یک مسئله بزرگ به مسائل کوچک تر، پیدا کردن علت ریشه ای، دیدن راه حل هایی که در نگاه اول واضح نیستند و مهم تر از همه، تشخیص این که اصلاً چه مسئله ای ارزش حل کردن دارد، مهارت هایی هستند که با قدرتمندتر شدن هوش مصنوعی ارزش بیش تری پیدا می کنند.

توانایی تجزیه مسائل پیچیده به مؤلفه های کوچک تر، عارضه یابی ریشه ای و اولویت بندی مسائل با اهمیت، با پیشرفت هوش مصنوعی ارزش مضاعفی می یابند.

کاهش هزینه و زمان آزمون ایده ها به واسطه هوش مصنوعی، باعث شده نقش مهندس ممتاز، از صرف یافتن سریع پاسخ، به طرح پرسش های دقیق تر و ارزیابی کم هزینه تر راه حل های مختلف تغییر یابد.

۴. تفکر انتقادی و ارزیابی

کیفی خروجی هوش مصنوعی

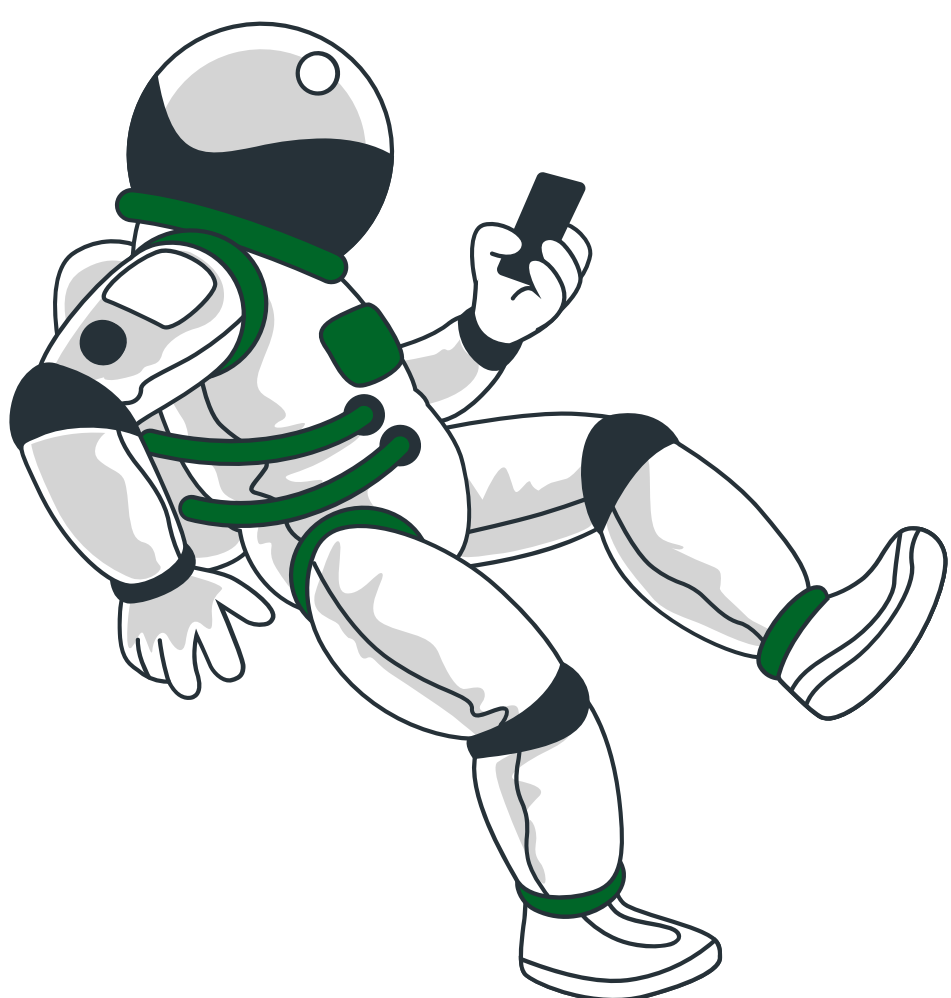
یکی از خطرناک ترین ویژگی های مدل های زبانی این است که می توانند با اعتماد به نفس کامل اشتباه کنند. کدی که از یک مدل دریافت می کنید ممکن است کامپایل شود، تست های اولیه را پاس کند و حتی در نگاه اول کاملاً منطقی به نظر برسد، اما در یک Edge Case خاص Race Condition (وضعیتی که در آن نتیجه اجرا به ترتیب زمانی اجرای هم زمان چند فرایند یا رشته بستگی دارد و خروجی را غیرقابل پیش بینی می کند) ایجاد کند، یک آسیب پذیری امنیتی داشته باشد یا زیر بار بالا تبدیل به یک گلوگاه شود.

یکی از تجربه‌هایی که احتمالاً بسیاری از برنامه‌نویسان هنگام کار با دستیارهای هوش مصنوعی داشته‌اند، این است که هرچه مسئله را دقیق‌تر تعریف کنید، خروجی بهتری دریافت می‌کنید. این اتفاق تصادفی نیست. بخش قابل‌توجهی از چیزی که امروز تحت عنوان Prompt Engineering یا گاهی Context Engineering درباره آن صحبت می‌کنیم، در عمل شباهت زیادی به همان مهارت قدیمی تحلیل درست مسئله دارد.

۹. معماری نرم‌افزار

اگر هوش مصنوعی را به یک تیم نرم‌افزاری تشبیه کنیم، هر روز در حال اضافه کردن برنامه‌نویسان سریع‌تر و ارزان‌تری به این تیم هستیم. اما افزایش تعداد برنامه‌نویسان لزوماً محصول بهتری ایجاد نمی‌کند. کسی باید تصمیم بگیرد این بخش‌ها چگونه کنار یک‌دیگر قرار بگیرند. مرز سرویس‌ها کجاست؟ داده متعلق به کدام بخش است؟ پیوستگی تا چه اندازه اهمیت دارد؟ سیستم چگونه مقیاس می‌شود؟ اگر یک وابستگی از دسترس خارج شود، چه اتفاقی می‌افتد؟ کجا باید Coupling را کاهش دهیم و کجا انجام این کار فقط پیچیدگی اضافه ایجاد می‌کند؟

هوش مصنوعی می‌تواند برای تمام این پرسش‌ها پاسخ تولید کند. حتی می‌تواند چند معماری مختلف پیشنهاد دهد و مزایا و معایب هرکدام را توضیح دهد. اما معماری نرم‌افزار مجموعه‌ای از Best Practice ها نیست که بتوان آن‌ها را بدون کانتکست روی هر سیستمی اعمال کرد. تقریباً تمام تصمیمات معماری Trade-off هستند. کارایی بهتر ممکن است سازگاری را کاهش دهد. دسترس‌پذیری بیش‌تر ممکن است پیچیدگی را افزایش دهد. جداسازی بیش‌تر بخش‌ها می‌تواند قابلیت استقرار را بهتر کند، اما خطایابی را سخت‌تر کند. امنیت بیش‌تر ممکن است روی کاربردپذیری اثر بگذارد.



هوش مصنوعی این مشکل را می‌تواند شدیدتر کند. چرا که هزینه تولید پیچیدگی را کاهش داده است. وقتی می‌توانید در چند دقیقه چند سرویس، message broker، cache، پایپ‌لاین CI و مجموعه‌ای از منیفست‌های Kubernetes تولید کنید، اضافه کردن پیچیدگی بسیار وسوسه‌کننده می‌شود، اما این که تولید یک چیز ساده شده است، به این معنی نیست که نگهداری آن نیز ساده شده است. مهندسی خوب یعنی بتوانید تشخیص دهید برای مسئله امروز چه مقدار مهندسی کافی است. نه کم‌تر از چیزی که سیستم نیاز دارد و نه بیش‌تر از آن (مشکلی به نام over-engineering).

۷. دیباگ

برگردیم به داستان ابتدای مقاله. اگر هوش مصنوعی می‌تواند باگی را که یک مهندس باتجربه صدها ساعت برای آن وقت گذاشته در چند ساعت پیدا کند، آیا مهارت دیباگ دیگر اهمیتی دارد؟

ابزارهای هوش مصنوعی توانایی فوق‌العاده‌ای در بررسی حجم بزرگی از کد، پیدا کردن الگوها، دنبال کردن مسیر اجرای برنامه و ارائه فرضیه دارند. اما برای استفاده مؤثر از این توانایی، باید بتوانید مسئله را برای آن‌ها قابل مشاهده کنید. چه لاگ‌هایی نیاز داریم؟ چه شاخص‌هایی باید جمع شوند؟ چگونه می‌توان باگ را بازتولید کرد؟ کدام رفتار سیستم غیرعادی است؟ از بین ده فرضیه‌ای که مدل زبانی ارائه کرده، کدام یک ارزش بررسی دارد؟

۸. تحلیل و طراحی سیستم‌ها

شاید تولید کد، یکی از بخش‌هایی باشد که بیش‌ترین تغییر را به واسطه هوش مصنوعی تجربه می‌کند، اما قبل از این که حتی یک خط کد نوشته شود، باید مجموعه‌ای از تصمیمات گرفته شوند. سیستم دقیقاً قرار است چه کاری انجام دهد؟ کاربران آن چه کسانی هستند؟ محدودیت‌های آن چیست؟ چه داده‌هایی وارد سیستم می‌شوند؟ چه فرایندهایی روی آن‌ها انجام می‌شوند؟ چه سناریوهایی ممکن است اتفاق بیفتند؟ اگر بخشی از سیستم از دسترس خارج شود، چه اتفاقی می‌افتد؟ اگر پاسخ این پرسش‌ها مشخص نباشد، سریع‌تر کد نوشتن لزوماً کمکی به شما نمی‌کند.

به همین دلیل یکی از مهم‌ترین مهارت‌های مهندس نرم‌افزار آینده توانایی گفتن «این خروجی خوب نیست» است. این موضوع ساده‌تر از چیزی که به نظر می‌رسد، نیست. برای این که بتوانید کیفیت خروجی یک مدل را ارزیابی کنید، خودتان باید درکی از کیفیت داشته باشید. اگر ندانید یک طراحی خوب چه ویژگی‌هایی دارد، چگونه می‌خواهید تشخیص دهید طراحی تولیدشده توسط هوش مصنوعی خوب است؟ اگر اصول امنیت را ندانید، چطور متوجه می‌شوید کدی که تولید شده آسیب‌پذیر است؟

۵. تست و مهندسی کیفیت

هرچه تولید کد ساده‌تر شود، تولید کد بیش‌تر نیز ساده‌تر می‌شود و کد بیش‌تر به معنی فضای بیش‌تر برای ایجاد خطاست. فرض کنید قبلاً یک توسعه‌دهنده در یک روز می‌توانست یک قابلیت کوچک را پیاده‌سازی کند. امروز همان توسعه‌دهنده با کمک دستیارها ممکن است چند قابلیت را در همان زمان پیاده‌سازی کند. سرعت توسعه چند برابر شده است، اما اگر فرایند کنترل کیفیت با همان سرعت رشد نکند، تنها چیزی که به دست آورده‌ایم، توانایی تولید سریع‌تر باگ است. به همین دلیل، تست در عصر هوش مصنوعی نه تنها اهمیت خود را از دست نمی‌دهد، بلکه احتمالاً مهم‌تر نیز خواهد شد.

البته منظور از تست صرفاً نوشتن چند Unit Test نیست. باید بدانید چه چیزی ارزش تست شدن دارد، کجا Integration Test نیاز است، چه زمانی End-to-End Test منطقی است، چه بخش‌هایی باید به شکل خودکار بررسی شوند و چه ویژگی‌هایی مانند کارایی، امنیت، قابلیت اطمینان و کاربردپذیری نیازمند ارزیابی جداگانه هستند.

۶. مهندسی به حد نیاز و کافی

یکی از جذاب‌ترین کارها برای مهندسان نرم‌افزار، حل مسائل پیچیده است. همین موضوع، گاهی باعث می‌شود مسئله‌ای که به یک راه حل ساده نیاز دارد را به یک مسئله بسیار پیچیده تبدیل کنیم. برای محصولی با چندصد کاربر میکروسرویس طراحی می‌کنیم؛ برای مشکلی که با یک صف ساده حل می‌شود، سراغ یک زیرساخت عظیم رویدادراشه می‌رویم و برای سیستمی که یک ماشین پاسخگوی نیاز آن است، از ابتدا به فکر Kubernetes می‌افتیم.

که هر هفته فریم‌ورک خود را عوض کنید یا هر ابزار جدید هوش مصنوعی را دنبال کنید. اتفاقاً بخش مهمی از آن، تشخیص تغییرات واقعی از موج‌های موقتی است. اما زمانی که یک تغییر واقعی اتفاق افتاد، مقاومت در برابر آن معمولاً تغییر را متوقف نمی‌کند؛ تنها باعث می‌شود دیگران زودتر از شما از آن استفاده کنند.

در انتها، به صحبت ابتدای این مقاله می‌رسیم. کسانی که با شرایط جدید سازگاری پیدا نکنند، از محیط حذف خواهند شد. احتمالاً عده‌ای هستند که تصور می‌کنند آن‌قدر برنامه‌نویس‌های خوبی هستند که این تطبیق‌پذیری نباید دغدغه آن‌ها باشد.

نرم‌افزار در فضاهای مبهم عمل می‌کنند تا آن محدوده‌ها را کشف و ایجاد کنند، با ذی‌نفعان تعامل می‌کنند تا نیازمندی‌ها و اولویت‌ها را تعیین کنند.

گلوگاه کیفیت و امنیت

هرچند هوش مصنوعی می‌تواند کد را با مقیاسی فراتر از توان انسان تولید کند، اما همچنان کنشگری ناقص است. ایمی جی کو -استاد و معاون آموزشی دانشکده اطلاعات دانشگاه واشنگتن- می‌گوید: «تخصص همیشه با دانش عمیق از کیفیت‌های نرم‌افزار و نحوه دستیابی به آن‌ها از طریق پیاده‌سازی مشخص می‌شود؛ درک پیچیدگی معماری؛ توانایی یادگیری و تغییر مستمر شیوه‌ها؛ ارائه اطلاعات معتبر، صادقانه و قابل اعتماد و دنباله بلندی از دیگر مهارت‌های نرم.» این اتکا به قضاوت انسانی، مجموعه جدیدی از گلوگاه‌ها ایجاد می‌کند. بیل نیکولز -سرپرست ابتکار اندازه‌گیری و آزمایش کاربردی در دانشگاه کارنگی ملون- گفت عامل محدودکننده در چرخه حیات نرم‌افزار از «آیا می‌توانیم بسازیمش؟» به «آیا می‌توانیم درکش کنیم، آزمونش کنیم و توجیهش کنیم؟» تغییر کرده است. او تمایز واضحی بین راستی‌آزمایی و اعتبارسنجی قائل می‌شود: راستی‌آزمایی (بررسی انطباق سیستم با مشخصات) می‌تواند خودکار شود، اما اعتبارسنجی (تصمیم‌گیری درباره این‌که آیا سیستم قابلیت‌های درست را برای محیط خود فراهم می‌کند) همچنان فرایندی انسانی و پر از قضاوت است. او می‌گوید بدون این نظارت، پایگاه‌های کد تولیدشده توسط ایجنت‌ها در معرض خطر تبدیل شدن به لجن غیرقابل نگهداری قرار دارند.

کاری که سال گذشته هنوز نیازمند چند ساعت فعالیت دستی بود، ممکن است امروز کاملاً قابل‌واگذاری به یک ایجنت باشد. حتی شیوه تعامل ما با محیط‌های توسعه در حال تغییر است. به همین دلیل، شاید مهم‌ترین توانایی یک مهندس نرم‌افزار در سال‌های آینده «یاد گرفتن» نباشد، بلکه «یاد گرفتن دوباره» باشد. باید بتوانید بعضی عادت‌هایی که سال‌ها با آن‌ها کار کرده‌اید را کنار بگذارید. باید حاضر باشید روند کار خود را تغییر دهید. باید ابزارهای جدید را امتحان کنید و در عین حال آن‌قدر هیجان‌زده نشوید که هر فناوری جدیدی را بدون ارزیابی وارد پروژه کنید. انطباق‌پذیری به این معنا نیست

خوب انجام شده، بهتر از آنچه خودم می‌توانستم انجام دهم، بدون نیاز به هیچ اصلاحی.» حالا مردم می‌پرسند اگر هوش مصنوعی اکنون می‌تواند نرم‌افزار را با هزینه کم‌تر بنویسد، اشکال‌زدایی کند و مستقر کند، تأثیر آن بر متخصصان انسانی توسعه نرم‌افزار چیست؟

از پیاده‌سازی به نظارت راهبردی

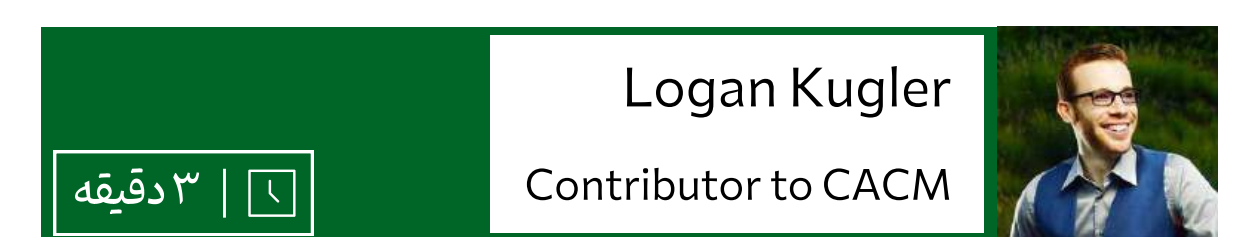
اجماع میان صاحب‌نظران این است که هرچند «کدنویس» ممکن است در معرض خطر باشد، اما «مهندس» همچنان ضروری می‌ماند، هرچند این نقش در حال بازتعریف شدن به‌عنوان ترکیبی از مدیر محصول، بازبین و هماهنگ‌کننده هوش مصنوعی است. جیمز آیورز -سرپرست گروه گردش‌های کاری هوش مصنوعی و نوسازی معماری در مؤسسه مهندسی نرم‌افزار دانشگاه کارنگی ملون- استدلال می‌کند که ارزش پیشنهادی یک مهندس حرفه‌ای همیشه بسیار فراتر از نحو زبان بوده است. آیورز گفت: «کدنویسی اغلب بخش آسان ماجراست. مهندسان نرم‌افزار بزرگ بر بخش‌های بسیار بیش‌تری از چرخه حیات توسعه نرم‌افزار تأثیر می‌گذارند تا فقط نوشتن کد. تحلیل نیازمندی‌ها، معماری و طراحی، استراتژی‌های آزمون مؤثر، برنامه‌ریزی و مدیریت ذی‌نفعان، همه این‌ها فعالیت‌هایی حیاتی برای موفقیت پروژه هستند.» آیورز یک مدل ذهنی پیشنهاد کرد که بین «کدنویس‌ها» و «مهندسان نرم‌افزار» تمایز قائل می‌شود. در این دیدگاه، کدنویس‌ها متخصصان زبان هستند که در محدوده‌هایی تعریف‌شده توسط دیگران فعالیت می‌کنند، مانند تیکت‌های Jira یا طراحی‌های مشخص. در مقابل، مهندسان

۱۰. انطباق‌پذیری با شرایط

و در نهایت به مهم‌ترین مهارت این لیست می‌رسیم؛ مهارتی که شاید تمام ۹ مورد قبلی نیز زیرمجموعه‌ای از آن باشند: «انطباق‌پذیری». اگر پنج سال پیش ابزارها و روش‌هایی را یاد گرفته بودید و تصمیم می‌گرفتید دیگر چیز جدیدی یاد نگیرید، شاید می‌توانستید برای مدت قابل‌توجهی با همان دانش به کار خود ادامه دهید. امروز فاصله میان تغییرات بسیار کم‌تر شده است. ابزاری که شش ماه پیش بهترین انتخاب شما برای یک کار بوده، ممکن است امروز جای خود را به ابزار دیگری داده باشد.

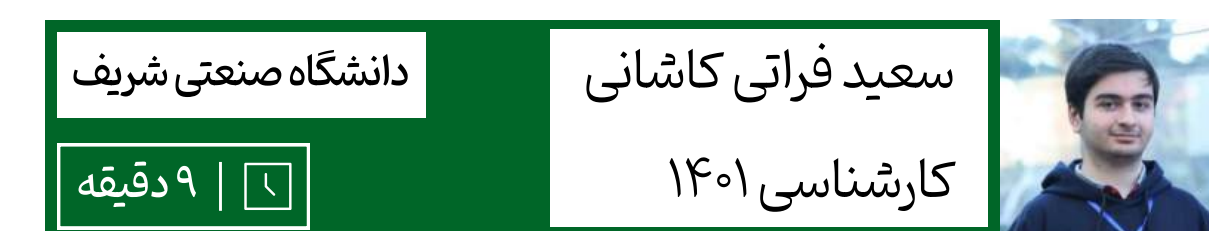
پایان برنامه‌نویس؟

از دستیارهای ساده کدنویسی تا مهندسان تمام‌عیار



در ژانویه ۲۰۲۶، داریو آمودی -مدیرعامل شرکت Anthropic- پیش‌بینی‌ای کرد که موجی از واکنش‌ها را در بخش فناوری ایجاد کرد: جهان ممکن است فقط شش تا دوازده ماه با مدل‌های هوش مصنوعی‌ای فاصله داشته باشد که قادرند همه وظایف مهندسی نرم‌افزار را به‌صورت سرتاسری انجام دهند. این جدول زمانی بسیار جسورانه بود، اما شواهدی از آن نیز نمایان بود. به‌عنوان مثال بوریس چرنی -رئیس بخش Claude Code در Anthropic- اعتراف کرد که ۱۰۰٪ کد خودش اکنون توسط هوش مصنوعی تولید می‌شود. برای چندین دهه، توسعه‌دهنده نرم‌افزار با توانایی ترجمه نیازهای انسانی به نحو دقیق و قابل خواندن برای ماشین تعریف می‌شد. اما انتشار پی‌اپی و سریع ابزارهای هوش مصنوعی در اوایل ۲۰۲۶، این تعریف را به نقطه شکست رسانده است. مت شومر -هم‌بنیان‌گذار Otherside AI- در مقاله‌ای که در اوایل ۲۰۲۶ در X ویرال شد، گفت: «من دیگر برای کار فنی واقعی شغلم لازم نیستم. من با زبان ساده توصیف می‌کنم که چه می‌خواهم ساخته شود و بعد فقط... ظاهر می‌شود. نه یک پیش‌نویس خام که نیاز به اصلاح داشته باشد؛ خود محصول نهایی. به هوش مصنوعی می‌گویم چه می‌خواهم، چهار ساعت از کامپیوترم دور می‌شوم، و برمی‌گردم و می‌بینم کار انجام شده؛

فناوری هراسی؛ امنیت یا ناآگاهی؟ از نوکیا پرسیم!



یک شخصیت مرموز وارد اتاق می‌شود. کت تیره پوشیده، پرده‌ها کشیده شده‌اند و قرار است حرف مهمی زده شود. پیش از شروع جلسه سُرّی! گوشی هوشمندش را خاموش می‌کند (تازه اگر داشته باشد)، شاید حتی باتری آن را کنار بگذارد و از جیب دیگر یک گوشی دکمه‌ای قدیمی بیرون بیاورد؛ چیزی شبیه همان نوکیاهایی که زمانی با یک بار شارژ چند روز دوام می‌آوردند. پیام فیلم روشن است: گوشی هوشمند یعنی ردیابی، شنود و خطر؛ و مهم‌تر از آن، گوشی ساده یعنی امنیت!

این تصویر فقط متعلق به سینما نیست. در دنیای واقعی هم بارها با همین منطق مواجه می‌شویم. کسانی که دوربین، GPS، سیستم‌عامل پیچیده، حساب کاربری ابری و ده‌ها برنامه نصب‌شده روی تلفن هوشمند را می‌بینند و نتیجه می‌گیرند بهترین راه امنیت، بازگشت به یک گوشی قدیمی است. در نگاه اول، استدلال بی‌راه به نظر نمی‌رسد. دستگاهی که اپلیکیشن ندارد، وای‌فای ندارد، بلوتوثش ابتدایی است و شاید حتی دوربین هم نداشته باشد، طبیعتاً بخشی از سطح حمله یک تلفن هوشمند را ندارد. اما امنیت با شمردن تعداد قابلیت‌ها تعیین نمی‌شود. گاهی همان دستگاه ساده، دقیقاً در بنیادی‌ترین بخش ارتباطش، یعنی شبکه‌ای که تماس و پیامک را حمل می‌کند، به فناوری‌ای متکی است که چند دهه از طراحی آن گذشته و بسیاری از فرض‌های امنیتی امروز را نداشته است.

این‌جاست که باید از نوکیا پرسیم: «واقعاً قدیمی‌تر بودن، امن‌تر بودن است؟» برای پاسخ، باید کمی به عقب برگردیم. شبکه‌های نسل دوم تلفن همراه یا 2G، به‌ویژه GSM^۱، محصول دورانی بودند که هدف اصلی شبکه موبایل، فراهم کردن تماس و پیامک دیجیتال در مقیاس بزرگ بود. در آن زمان، مدل تهدید امروز وجود نداشت؛ تجهیزات مخابراتی بسیار گران‌تر و دسترسی به آن‌ها محدودتر بود و تصور این‌که یک مهاجم بتواند با سخت‌افزار نسبتاً در دسترس، نقش

بخشی از شبکه اپراتور را تقلید کند، به اندازه امروز جدی نبود.

یکی از مهم‌ترین ضعف‌های معماری GSM همین‌جاست: احراز هویت یک‌طرفه. شبکه از سیم‌کارت می‌خواهد ثابت کند مشترک معتبری است، اما گوشی الزاماً به همان شکل از شبکه نمی‌خواهد ثابت کند که واقعاً شبکه اپراتور است. به زبان ساده، گوشی باید خودش را به شبکه معرفی کند، ولی شبکه در 2G همان سطح از «من واقعاً همان کسی هستم که ادعا می‌کنم» را به گوشی ارائه نمی‌دهد. این تفاوت کوچک روی کاغذ، در عمل بسیار بزرگ است.

نتیجه، امکان ساخت «ایستگاه پایه جعلی» است؛ سامانه‌ای که خودش را شبیه دکل مخابراتی نشان می‌دهد و تلاش می‌کند تلفن‌های اطراف را به اتصال وادار کند. ابزارهایی که با نام‌هایی مانند IMSI Catcher^۲ یا در برخی کاربردها Stingray شناخته می‌شوند، دقیقاً از چنین ایده‌ای بهره می‌برند. اگر گوشی به شبکه جعلی اعتماد کند، مهاجم می‌تواند اطلاعاتی درباره مشترک به دست آورد و بسته به پیاده‌سازی و شرایط، زمینه را برای رهگیری، کاهش امنیت ارتباط یا حملات دیگر فراهم کند. GSMA^۳ نیز نبود احراز هویت متقابل در 2G را یکی از دلایل امکان سوء استفاده از ایستگاه‌های پایه جعلی می‌داند.

مشکل فقط به «جعلی بودن دکل» محدود نمی‌شود. الگوریتم‌های رمزنگاری نسل دوم نیز فرزند زمان خود هستند. خانواده A5 که برای تأمین محرمانگی ارتباط رادیویی GSM طراحی شده بود، امروز دیگر معیار مناسبی برای یک ارتباط حساس به‌شمار نمی‌رود. برای نمونه، A5/1 با وجود آن‌که نسخه قوی‌تر این خانواده محسوب می‌شد، به دلیل فضای کلید محدود و ضعف‌های ساختاری در برابر حملات شناخته‌شده‌ای مانند Known-Plaintext و به‌ویژه حملات مبتنی بر Time-Memory Trade-off^۴ آسیب‌پذیر است؛ حملاتی که با پیش‌محاسبه و ذخیره حجم بزرگی از داده، می‌توانند زمان لازم برای بازیابی کلید را به‌طور چشمگیری کاهش دهند. A5/2 نیز اساساً به‌عنوان نسخه‌ای ضعیف‌تر طراحی شده بود و در برابر حملات مبتنی بر متن معلوم مقاومت بسیار کم‌تری دارد. از همین رو، نهادهایی مانند ENISA استفاده از این الگوریتم‌های قدیمی را برای ارتباطات حساس مناسب نمی‌دانند. مهم‌تر این‌که معماری GSM امکان استفاده از

الگوریتم‌های مختلف و حتی حالت بدون رمزنگاری را نیز پیش‌بینی کرده است. در نتیجه، همان گوشی‌ای که به خاطر «سادگی» انتخاب شده، ممکن است از نظر امنیت لایه رادیویی، روی زمینی بازی کند که قواعد آن متعلق به چند دهه قبل است.

حمله مهم دیگر، تنزل نسل یا Downgrade است. تلفن‌های همراه برای حفظ پوشش شبکه، معمولاً توانایی جابه‌جایی میان نسل‌های مختلف را دارند. یک گوشی 4G یا 5G ممکن است در شرایطی که شبکه جدیدتر در دسترس نیست، به 3G یا 2G برگردد. مهاجم می‌تواند تلاش کند دستگاه را متقاعد کند که شبکه امن‌تر در دسترس نیست و آن را به 2G بکشاند؛ جایی که ضعف‌های قدیمی دوباره فعال می‌شوند. به همین دلیل، حتی داشتن یک تلفن هوشمند جدید نیز اگر همچنان اجازه اتصال به 2G را بدهد، لزوماً به معنای حذف کامل این ریسک نیست. مستندات امنیتی اندروید صراحتاً توضیح می‌دهند که یکی از کاربردهای قابلیت «غیرفعال کردن 2G» جلوگیری از همین سناریوهای ایستگاه پایه جعلی و Downgrade است.

پس نسل‌های جدید چه چیزی را بهتر کرده‌اند؟ در 4G/LTE، یکی از تغییرات بنیادی، احراز هویت متقابل است. در سازوکار AKA^۵، نه تنها مشترک باید هویت معتبر خود را برای شبکه اثبات کند، بلکه دستگاه نیز می‌تواند اصالت شبکه را بررسی کند. این همان قطعه‌ای است که در مدل کلاسیک 2G وجود نداشت. NIST^۶ در بررسی امنیت LTE، احراز هویت متقابل، جداسازی بهتر کلیدهای رمزنگاری و حفاظت از تمامیت پیام‌های کنترلی را از پیشرفت‌های مهم امنیتی 4G می‌داند.

در LTE همچنین معماری مدیریت کلید بسیار منظم‌تر و الگوریتم‌های رمزنگاری مدرن‌تری وارد شدند. علاوه بر محرمانگی، «تمامیت» پیام‌های سیگنالینگ اهمیت بیش‌تری پیدا کرد؛ یعنی شبکه و دستگاه فقط نمی‌خواهند کسی پیام را نخواند، بلکه می‌خواهند مطمئن شوند پیام در میانه راه دستکاری نشده است. این تفاوت مهمی است. امنیت فقط قفل کردن محتوا نیست؛ تشخیص دستکاری فرمان‌های شبکه، پیام‌های کنترلی و فرایندهای احراز هویت نیز بخشی از مسئله است.

یک گام دیگر هم جلو می‌رود. در نسل‌های 5G

هوشمند ترسناک» گاهی به کاربر امکان دیدن و قطع کردن خطری را می‌دهد که یک گوشی قدیمی اصلاً رابطی برای نمایش آن ندارد.

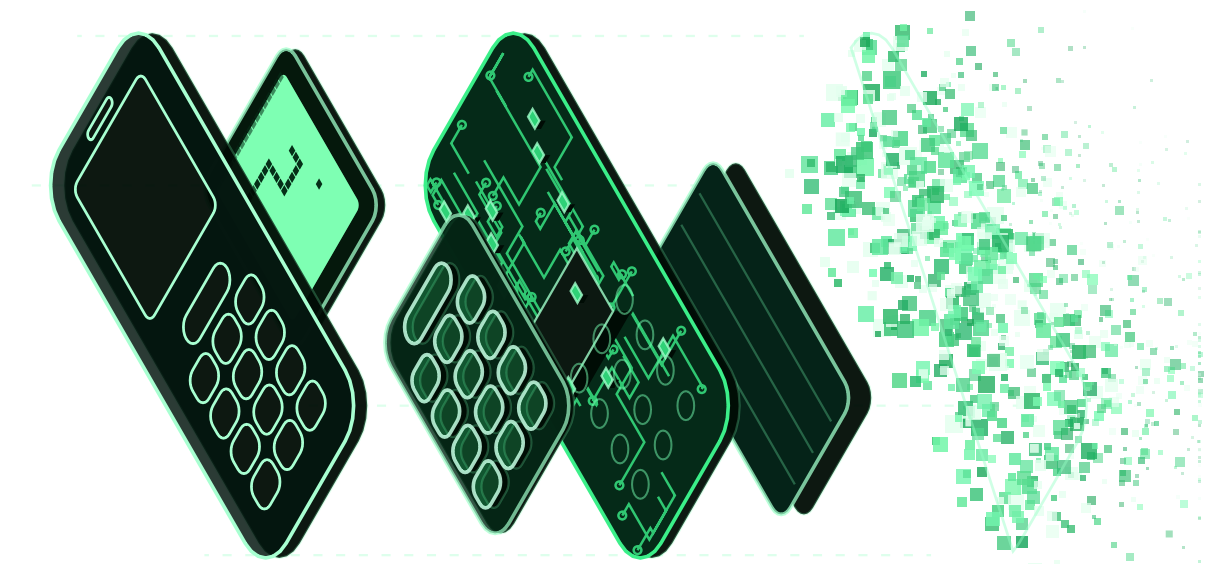
شاید به همین دلیل، بهتر باشد در فیلم بعدی، وقتی شخصیت امنیتی داستان یک نوکیای قدیمی از جیبش بیرون می‌آورد، کمی شک کنیم. شاید او واقعاً با حذف دوربین، اپلیکیشن‌ها و حساب‌های ابری، بخشی از سطح حمله را کم کرده باشد؛ اما اگر آن گوشی فقط GSM را پشتیبانی کند، هم‌زمان بخشی از دفاع شبکه‌ای را نیز کنار گذاشته است. امنیت همیشه جمع کردن قابلیت‌ها نیست؛ گاهی انتخاب میان دو مجموعه متفاوت از ریسک‌هاست.

مشکل اصلی از جایی شروع می‌شود که «پیچیده» را مترادف «ناامن» و «قدیمی» را مترادف «قابل اعتماد» می‌گیریم. فناوری جدید می‌تواند آسیب‌پذیری‌های جدید بسازد، اما هم‌زمان نتیجه چند دهه تجربه، حمله، پژوهش و اصلاح استانداردها نیز هست. بسیاری از سازوکارهایی که امروز در 4G و 5G بدیهی به نظر می‌رسند، دقیقاً به این دلیل اضافه شده‌اند که نسل‌های قبل در برابر سناریوهای واقعی ضعف نشان داده‌اند.

پس شاید پرسش درست این نباشد که «گوشی هوشمند امن‌تر است یا نوکیا؟» پرسش درست این است: از چه چیزی می‌ترسیم، مهاجم ما چه توانایی‌ای دارد و کدام فناوری دقیقاً در برابر آن تهدید دفاع بهتری ارائه می‌دهد؟

امنیت با نوستالژی ساخته نمی‌شود؛ با شناخت ساخته می‌شود. گاهی خاموش کردن یک قابلیت واقعاً امنیت را بیش‌تر می‌کند، گاهی استفاده از یک دستگاه ساده تصمیم هوشمندانه‌ای است و گاهی نیز همان «سادگی» ما را به استنادی برمی‌گرداند که مهاجمان سال‌ها فرصت مطالعه و سوء استفاده از آن را داشته‌اند. دشمن اصلی نه تلفن هوشمند است، نه گوشی دکمه‌ای. دشمن اصلی تصمیمی است که فقط از ظاهر فناوری گرفته شود.

اگر قرار است از چیزی بترسیم، بهتر است اول آن را بفهمیم.



این البته دفاع بی‌قیدوشرط از تلفن هوشمند نیست. یک گوشی هوشمند واقعاً سطح حمله بزرگی دارد: سیستم‌عامل، مرورگر، پردازنده بیس‌بند، وای‌فای، بلوتوث، NFC، ده‌ها حسگر، اپلیکیشن‌های شخص ثالث و حساب‌های ابری، همگی می‌توانند در مدل‌های تهدید مختلف مهم باشند. اگر دستگاه به‌روزرسانی امنیتی دریافت نکند، برنامه‌های نامطمئن روی آن نصب شوند یا کاربر مجوزها را بدون دقت بدهد، داشتن 5G به‌تنهایی او را نجات نمی‌دهد. از آن طرف، یک گوشی ساده جدید که 4G و VoLTE^۵ دارد و نرم‌افزارش حداقلی است، ممکن است برای بعضی کاربردها انتخابی بسیار منطقی باشد. بنابراین مسئله «هوشمند یا دکمه‌ای» نیست؛ مسئله این است که کدام دستگاه، با کدام شبکه، در برابر کدام مهاجم و برای حفاظت از کدام دارایی استفاده می‌شود.

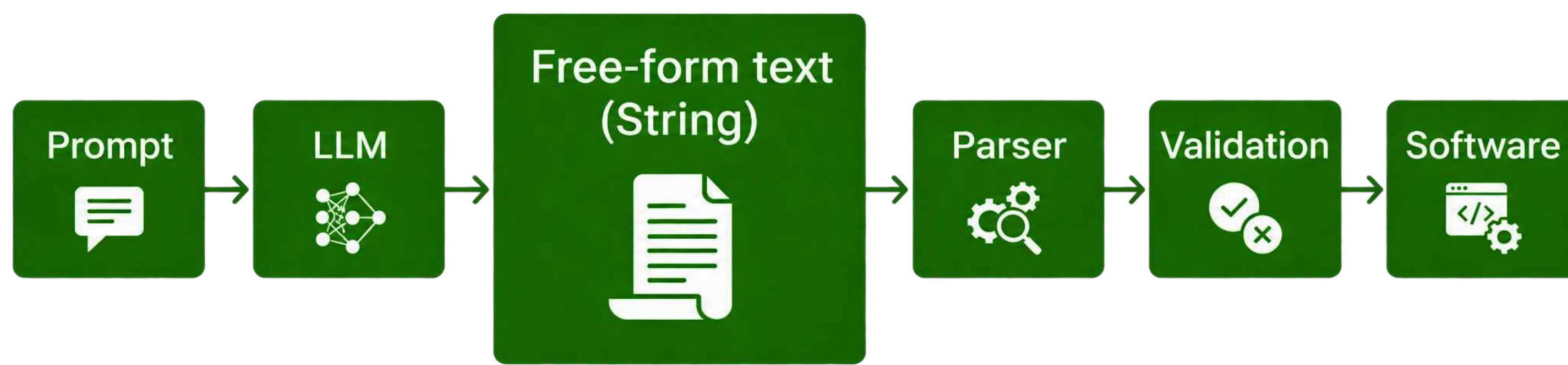
در امنیت، به این شیوه فکر کردن مدل تهدید می‌گویند. اگر نگرانی شما نصب بدافزار از طریق اپلیکیشن‌هاست، کاهش قابلیت‌های دستگاه می‌تواند مفید باشد. اگر نگرانی شما رهگیری تماس و پیامک روی شبکه سلولی است، یک گوشی 2G قدیمی ممکن است انتخاب بدتری باشد. اگر هدف شما محرمانگی محتوای مکالمه است، پیام‌رسانی با رمزنگاری سرتاسری روی یک تلفن به‌روز می‌تواند مزیتی داشته باشد که تماس GSM کلاسیک اساساً برای ارائه آن طراحی نشده است؛ استانداردهای GSM نیز حفاظت انتهایی را خارج از محدوده امنیت خود تعریف می‌کردند. اگر نگرانی شما مکان‌یابی و فراداده است، حتی رمزنگاری قوی محتوای پیام نیز همه مسئله را حل نمی‌کند، چون شبکه برای ارائه سرویس هنوز ناچار است اطلاعاتی درباره حضور و جابه‌جایی مشترک بداند.

نکته جالب این است که خود تلفن‌های هوشمند جدید، برخلاف تصویری که گاهی از آن‌ها ساخته می‌شود، ابزارهایی برای کاهش بخشی از این ریسک‌ها اضافه کرده‌اند. در اندروید، روی سخت‌افزارهای پشتیبانی‌شده می‌توان اتصال 2G را در سطح رادیویی غیرفعال کرد. در Android 16 نیز بخش «امنیت شبکه موبایل» برای دستگاه‌های سازگار قابلیت‌هایی مانند هشدار هنگام اتصال به شبکه بدون رمزنگاری یا تبادل شناسه‌هایی مانند IMSI و IMEI^۶ دارد. یعنی همان «گوشی

قبلی شناسه دائمی مشترک، مانند IMSI، در برخی فرایندهای سیگنالینگ می‌توانست به صورت plaintext روی رابط رادیویی ارسال شود و در نتیجه امکان شناسایی یا ردیابی مشترک را فراهم کند. در 5G برای کاهش این ریسک، مفهوم SUCI^۱ معرفی شده است؛ شناسه‌ای موقت و پوشیده‌شده که از شناسه دائمی مشترک ساخته می‌شود. در سازوکار استاندارد، گوشی با استفاده از طرح‌های رمزنگاری مبتنی بر ECC^۲ و کلید عمومی شبکه اصلی اپراتور^۳، شناسه دائمی را پنهان می‌کند و به جای ارسال مستقیم IMSI/SUPI^۴، مقدار SUCI را برای شبکه می‌فرستد. به این ترتیب، حتی شنود رابط رادیویی نیز در حالت عادی نباید به افشای مستقیم هویت دائمی مشترک منجر شود. این تغییر نشان می‌دهد که در طراحی 5G، حفاظت از هویت مشترک نیز از ابتدا بخشی از مدل امنیتی در نظر گرفته شده است، نه این که امنیت صرفاً به رمزنگاری محتوای ارتباط محدود شود.

البته این‌جا نباید به دام افسانه دیگری بیفتیم: «پس 5G امن است و تمام!» هیچ شبکه‌ای چنین تضمینی ندارد. 4G و 5G همچنان حملات، ضعف‌های پیاده‌سازی، آسیب‌پذیری‌های مودم، خطاهای اپراتور، حملات محروم‌سازی از سرویس و مسائل حریم خصوصی خود را دارند. علاوه بر آن، اگر یک گوشی جدید اجازه داشته باشد برای سازگاری به 2G برگردد، مهاجم ممکن است اصلاً تلاش نکند 5G را بشکند؛ کافی است قربانی را از بزرگراه جدید به جاده قدیمی هدایت کند. حتی راهنماهای امنیتی GSMA برای 5G نیز بر محدود کردن تنزل از 5G به نسل‌های قدیمی‌تر، به خصوص 2G، تأکید می‌کنند.

و این دقیقاً همان جایی است که «فناوری‌هراسی» می‌تواند ما را به تصمیم اشتباه برساند. ما معمولاً چیزی را که می‌بینیم خطرناک‌تر از چیزی می‌دانیم که نمی‌بینیم. در تلفن هوشمند، دوربین را می‌بینیم، میکروفن را می‌بینیم، مجوزهای برنامه‌ها را می‌بینیم، GPS را می‌شناسیم و درباره بدافزار و جاسوس‌افزار شنیده‌ایم. بنابراین خطر آن برایمان ملموس است. اما پروتکل سیگنالینگ، احراز هویت شبکه، الگوریتم رمزنگاری لایه رادیویی یا سازوکار مدیریت کلید در یک گوشی دکمه‌ای را نمی‌بینیم. گوشی ساده آرام و بی‌آزار به نظر می‌رسد، چون پیچیدگی‌های پشت صفحه‌ای کوچک و چند دکمه پنهان شده است.



حرف‌زن؛ تصمیم بگیر!

تولید متن تمام شد؛ نوبت تصمیم است.



دانشگاه صنعتی شریف

معین آعلی

۲۰ دقیقه

کارشناسی ۱۴۰۱

هر گزینه چقدر است؟» فضای پاسخ از قبل بسته است؛ مدل فقط باید آن را با احتمال پر کند.

TypeSafe می‌گوید این زنجیره را می‌شود از پایه بازطراحی کرد: اگر مسئله یک تصمیم است، چرا مدلی نسازیم که از اول برای تولید انتخاب طراحی شده، نه برای تولید رشته متنی؟

System One: مدلی که تصمیم تولید می‌کند، نه متن

نام System One Models از تمایز معروف دنیل کانمان میان System 1 و System 2 در کتاب Thinking, Fast and Slow الهام گرفته شده است. کانمان در این کتاب ذهن انسان را به دو شیوه تفکر تقسیم می‌کند: System ۱ سریع، خودکار و شهودی است و تقریباً بدون تلاش آگاهانه کار می‌کند؛ مثلاً وقتی چهره عصبانی کسی را تشخیص می‌دهید یا حاصل ۲+۲ را می‌گویید. System 2 کند، آگاهانه و تحلیلی است و برای کارهایی که تمرکز می‌خواهند فعال می‌شود؛ مثلاً حل یک مسئله ریاضی پیچیده یا مقایسه دقیق دو گزینه. TypeSafe همین ایده را برای ماشین بازتعریف کرده: مدلی سریع، ارزان و بدون استدلال زنجیره‌ای، برای تصمیم‌های محدود و پرتکرار داخل نرم‌افزار. به این خانواده از مدل‌ها دو چیز می‌دهید: یک state (محتوایی که باید درباره‌اش تصمیم گرفته شود؛ مثل متن یک تیکت، یک مقاله یا یک تراکنش) و مجموعه‌ای از questions که در کد خودتان تعریف می‌کنید. سه primitive این پرسش‌ها را می‌سازند:

۱. پرسش choice: از میان چند گزینه از پیش تعریف‌شده انتخاب می‌کند و توزیع احتمال روی همه گزینه‌ها را برمی‌گرداند.
۲. پرسش score: روی یک مقیاس ترتیبی (حداکثر ۱۰ سطح) امتیاز می‌دهد و میانگین وزنی سطح‌ها را برمی‌گرداند
۳. پرسش noul (مخفف Bernoulli): به یک گزاره بله/خیر، با عددی بین ۰ و ۱ جواب می‌دهد؛ احتمال درست بودن آن گزاره.

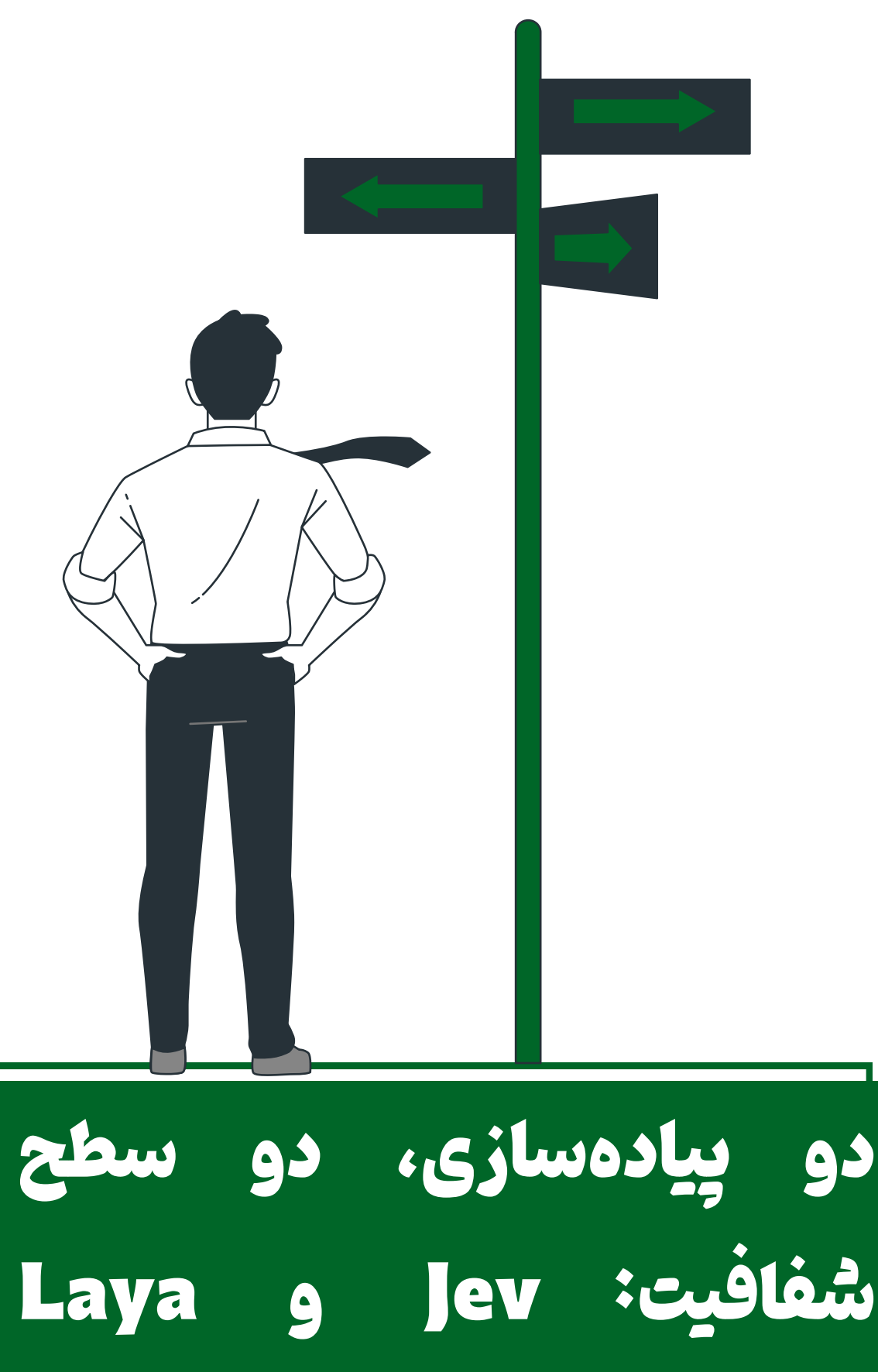
نکته مهم این‌جاست که شما دیگر از مدل نمی‌پرسید «نظرت درباره این متن چیست؟»؛ می‌پرسید: «برای این پرسش مشخص، احتمال

مدل‌های زبانی بزرگ در چند سال گذشته یک کار را بهتر از هر زمان دیگری انجام داده‌اند: حرف زدن. مقاله می‌نویسند، کد تولید می‌کنند، به ایمیل جواب می‌دهند، استدلال می‌کنند. اما یک پرسش مهندسی ساده معمولاً زیر همه این هیجان گم می‌شود: اگر قرار نیست خروجی را انسانی بخواند، اصلاً چرا مدل باید یک رشته متنی تولید کند؟ یک تیکت پشتیبانی باید به کدام تیم برود؟ یک تراکنش fraud است یا نه؟ یک مقاله به چه هشتک‌هایی تعلق دارد؟ در همه این‌ها، نرم‌افزار به یک پاراگراف قانع‌کننده نیاز ندارد؛ فقط یک تصمیم می‌خواهد، بله یا خیر! همین. نه توضیح، نه مقدمه، نه «به نظر می‌رسد که...».

از دل همین مشاهده، شرکت TypeSafe ایده System One Models را مطرح کرده و اولین محصول این خانواده، مدلی به نام Jev، در ۱۵ سپتامبر ۲۰۲۶ به صورت early access روی OpenRouter عرضه شده است. ادعای اصلی TypeSafe این است: برای بخش بزرگی از کارهایی که امروز با LLM انجام می‌دهیم، تولید متن لازم نیست و یک تصمیم ساختاریافته و کالبره‌شده کافی است. از منظر مهندسی، این را می‌توان تغییری در رابط مدل دانست؛ به این معنا که مدل از تولید مستقیم متن، به تصمیم‌گیری احتمالی بر مبنای وضعیت موجود حرکت می‌کند. همین تغییر در رابط، محور اصلی چیزی است که در ادامه بررسی می‌کنیم.

چرا هوش مصنوعی باید حرف‌بزند؟

مدل‌های زبانی برای تولید دنباله توکن ساخته شده‌اند: ورودی هر چیزی می‌تواند باشد، خروجی هم از یک کلمه تا هزاران خط کد. همین آزادی، بزرگ‌ترین نقطه قوت آن‌هاست؛ اما وقتی مسئله فقط یک طبقه‌بندی یا یک تصمیم دوتایی است، همین آزادی به یک زنجیره اضافه تبدیل می‌شود:



دو پیاده‌سازی، دو سطح شفافیت: Jev و Laya

ایده System One محصول یک شرکت نیست. TypeSafe در ۱۵ سپتامبر ۲۰۲۶ نسخه early access مدل Jev را منتشر کرد؛ فقط سه روز بعد، در ۱۸ سپتامبر، Convai Innovations مدل Laya را به صورت open-weight منتشر کرد. این فاصله کوتاه به این معنا نیست که Laya واکنشی به Jev بوده است؛ طبق توضیح نویسنده Laya، کار روی این ایده از مارس ۲۰۲۵ شروع شده و در دو مقاله arXiv در سال ۲۰۲۵ آمده بود؛ یعنی بسیار پیش از معرفی Jev. هم‌زمانی این دو انتشار بیش‌تر نشان می‌دهد System One یک محصول تک‌شرکتی نیست و چند تیم مستقل از هم به سمت همین رابط -یعنی تصمیم typed و احتمالاتی، نه متن- حرکت کرده‌اند. TypeSafe با Jev یک سرویس مدیریت‌شده و بسته ارائه کرده؛ اما Laya همان ایده را به صورت open-weight پیاده کرده و مهم‌تر از آن، معماری‌اش را کامل مستند کرده است. همین تفاوت شفافیت باعث می‌شود بهترین راه برای فهم چگونگی کار این مدل‌ها، نگاه کردن به Laya باشد، نه Jev.

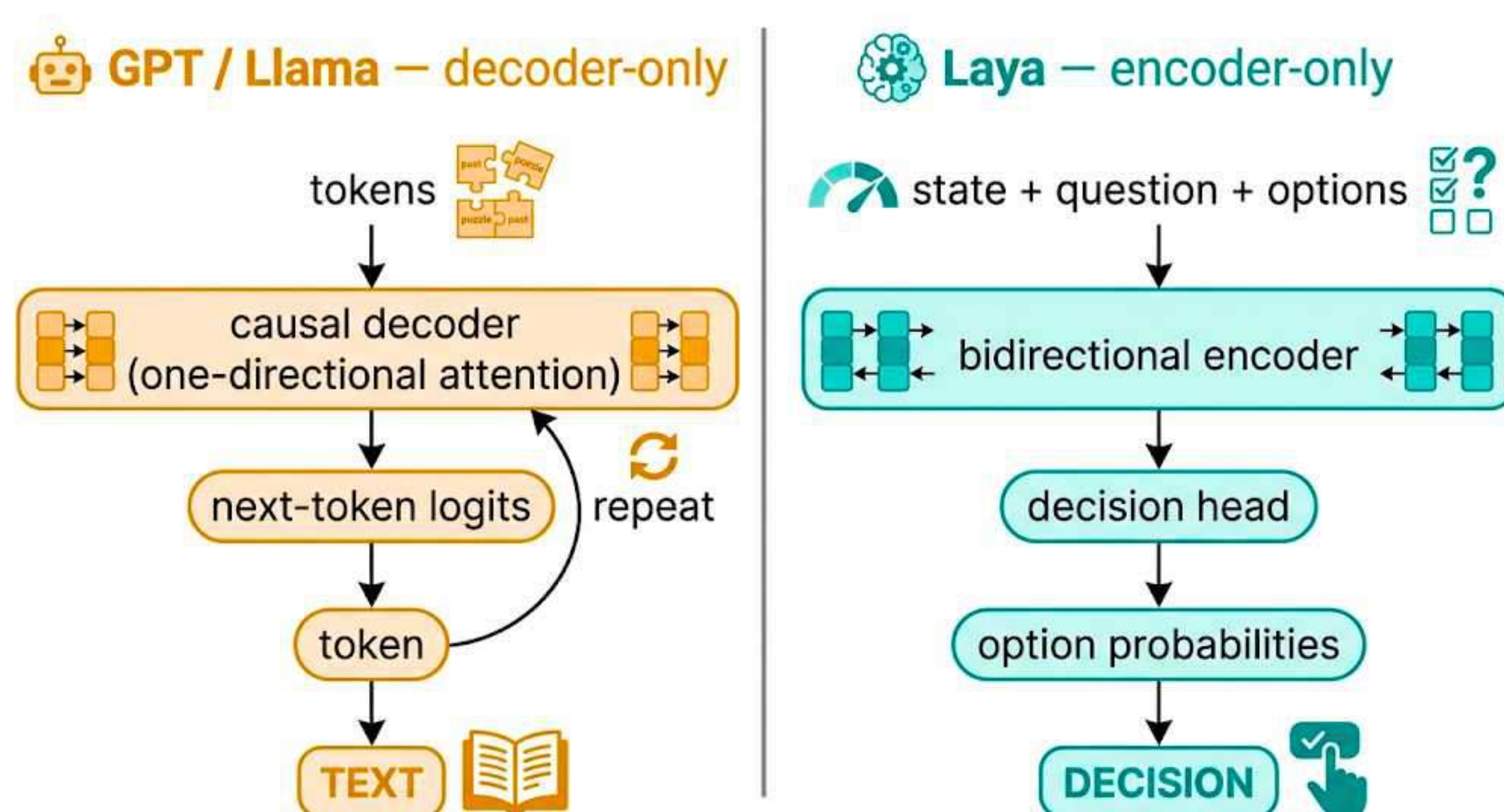
در معماری ای مثل Laya، چون فضای هر پاسخ از پیش بسته است (چند گزینه ثابت، یا یک بازه عددی محدود)، این تکرار ترتیبی اصلاً وجود ندارد: یک عبور رو به جلو روی انکودر کافی است تا امتیاز همه گزینه‌های همه پرسش‌های یک درخواست به دست بیاید. Jev هم تا جایی که از توضیحات عمومی TypeSafe برمی‌آید، روی همین اصل بنا شده: پرهیز از تولید ترتیبی و تمرکز بر محاسبه موازی روی یک فضای خروجی محدود.

لایه دوم تفاوت به مرحله آموزش برمی‌گردد. آموزش یک LLM معمولاً دو مرحله دارد. اول pre-training: مدل روی حجم عظیمی از متن یاد می‌گیرد توکن بعدی را پیش‌بینی کند. بعد post-training: مدل با روش‌هایی مثل RLHF^۳ به سمت پاسخ‌هایی هدایت می‌شود که ارزیاب‌های انسانی ترجیح می‌دهند. تفاوت Jev در همین مرحله دوم است. TypeSafe برای Jev از روشی به نام RLCD^۴ نام می‌برد که نه به «پاسخی که خوشایندتر است»، بلکه به «احتمالی که صادقانه گزارش شده» پاداش می‌دهد. Laya هم روش خود را RLCD می‌نامد و جزئیات بیش‌تری منتشر کرده است: مدل یک توزیع احتمال گزارش می‌دهد و پاداشش با یک «قاعده امتیازدهی اکیداً درست» محاسبه می‌شود. این نوع قاعده‌ها طوری ساخته شده‌اند که بهترین راه برای گرفتن بیش‌ترین پاداش، گزارش دقیقاً همان احتمالی است که مدل واقعاً به آن باور دارد؛ اغراق در اطمینان یا کم‌گویی عمدی، در میانگین، پاداش کم‌تری می‌گیرد.

یک مثال ساده این را روشن می‌کند. فرض کنید مدلی روی ۱۰۰ تیکت مشابه هر بار با اطمینان ۹۹٪ می‌گوید «باگ است»، ولی فقط ۷۰٪ تا از آن‌ها واقعاً باگ بوده‌اند. این مدل کالیبره نیست. مدلی که برای همان تیکت‌ها ۷۰٪ گزارش می‌کند، با نرخ واقعی هم‌خوان است و یک «قاعده امتیازدهی اکیداً درست» در میانگین به آن پاداش بیش‌تری می‌دهد. توجه کنید که این مقایسه فقط روی تعداد زیادی مورد معنا دارد؛ در یک مورد منفرد، پاسخ مطمئن‌تر ممکن است درست باشد و امتیاز بالاتری هم بگیرد. به همین دلیل هدف آموزشی فقط «آیا جواب درست بود؟» نیست؛ این هم هست که «آیا احتمالی که گزارش دادی با نرخ واقعی درستی‌اش سازگار بود؟». این ویژگی را کالیبره بودن می‌گویند: وقتی مدل

معماری Laya: یک encoder، نه یک decoder

طبق مستندات Laya، backbone نسخه انگلیسی این مدل ModernBERT-large با حدود ۳۹۵ میلیون پارامتر است - یک ترنسفورمر دوطرفه، از همان خانواده BERT^۱، نه از خانواده GPT^۲. روی این backbone، یک «سر انتخاب» اختصاصی نشسته: دو لایه ترنسفورمر اضافه، یک «option-marker scorer» و یک «act/escalate head»، که مجموعاً پارامترهای مدل را به حدود ۴۲۱ میلیون می‌رسانند. مکانیزم کار به این شکل است: هر گزینه پرسش در ورودی مدل یک نشانه مخصوص به خودش می‌گیرد. مدل روی کل دنباله (state + question + options) یک بار عبور می‌کند، نمایش هر marker را می‌گیرد، «سر انتخاب» روی آن یک امتیاز می‌سازد و در پایان یک softmax (تابعی که مجموعه‌ای از امتیازها را به توزیع احتمالی با مجموع یک تبدیل می‌کند) روی امتیاز همه گزینه‌های همان پرسش زده می‌شود. این معماری را بهتر می‌شود با کنار هم گذاشتن با یک LLM مولد فهمید.



این‌جا یک سوء برداشت رایج باید روشن شود: آیا Laya فقط یک LLM است که دیکودرش را برداشته‌اند؟ خیر. حذف دیکودر از یک مدل مولد به‌تنهایی یک مدل انتخابی نمی‌سازد. چیزی که Laya را برای تصمیم‌گیری مؤثر می‌کند، مجموعه‌ای از انتخاب‌های طراحی است: هدف آموزش، ساختار «سر انتخاب»، نمایش پرسش و گزینه‌ها و روش inference. همه از ابتدا برای امتیازدهی به گزینه‌ها طراحی شده‌اند، نه برای تولید توکن بعدی. توجه دوطرفه یک انکودر به مدل اجازه می‌دهد هم‌زمان به کل state و همه گزینه‌ها نگاه کند؛ در توجه علی یک دیکودر، هر توکن فقط توکن‌های قبل از خودش را می‌بیند، پس مثلاً state نمی‌تواند گزینه‌هایی را ببیند که بعد از آن در ورودی آمده‌اند.

در مقابل، TypeSafe درباره معماری داخلی Jev بسیار کم‌تر گفته است. آن‌چه رسماً اعلام شده، «sampler موازی» و ساختار non-autoregressive برای تصمیم‌های typed است، نه یک نمودار معماری. بنابراین وقتی می‌گوییم Jev همه پاسخ‌های یک درخواست را هم‌زمان محاسبه می‌کند، این یک توصیف مفهومی از رفتار بیرونی مدل است، نه ادعایی درباره جزئیات معماری‌اش.

چرا این ساختار سریع‌تر است؟

برای فهم این تفاوت، باید دقیق‌تر از token به token صحبت کرد. در یک مدل autoregressive، تولید هر توکن جدید به یک محاسبه جدید وابسته است و این محاسبات به‌صورت ترتیبی انجام می‌شوند. در رمزگشایی استاندارد، توکن دوم را نمی‌شود پیش از تعیین توکن اول تولید کرد. تکنیک‌هایی مثل KV cache (حافظه میانی که کلیدها و مقادیر محاسبه‌شده توکن‌های قبلی را نگه می‌دارد تا دوباره محاسبه نشوند) باعث می‌شوند در هر مرحله کل context قبلی از صفر دوباره از میان لایه‌های ترنسفورمر عبور داده نشود، اما همچنان وابستگی ترتیبی میان توکن‌ها باقی می‌ماند: برای یک پاسخ n-توکنی، باید n مرحله متوالی طی شود. (روش‌هایی مثل speculative decoding این هزینه را تا حدی کم می‌کنند، اما وابستگی ترتیبی را از بین نمی‌برند.)

نکته مهم‌تر از رقم دقیق، ساختار قیمت‌گذاری است: وقتی خروجی یک مدل چند بایت است نه چند پاراگراف، مدل قیمت‌گذاری «به‌ازای توکن خروجی» دیگر معنای اقتصادی زیادی ندارد و این، همان چیزی است که این خانواده از مدل‌ها را جدا از بحث سرعت، از نظر هزینه در کلاس دیگری نسبت به LLM های عمومی قرار می‌دهد.

یک نمونه: مسیریابی یک تیکت پشتیبانی

در این بخش، شکل واقعی یک فراخوانی Jev را می‌بینیم؛ ساختار درخواست و پاسخ مطابق مستندات OpenRouter است. فرض کنید متن یک تیکت پشتیبانی این است: «صفحه پرداخت من بعد از زدن دکمه Pay خالی می‌ماند و در دو مرورگر امتحان کرده‌ام.» دو پرسش مطرح می‌کنیم: آیا مشتری باگ گزارش کرده است یا نه (noul) و این تیکت باید به کدام تیم برود (choice).

```
curl https://openrouter.ai/api/alpha/decisions \
-H "Authorization: Bearer $OPENROUTER_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "typesafe/jev-1.13",
  "state": "My checkout page shows a blank screen after I click Pay. I have tried two browsers.",
  "questions": {
    "is_bug": {
      "type": "noul",
      "instructions": "Is the customer reporting a software defect?",
      "criteria": {
        "true": "The customer describes broken or unexpected product behavior.",
        "false": "The customer is asking a question or requesting a feature."
      }
    }
  },
  "team": {
    "type": "choice",
    "instructions": "Which team should own this ticket?",
    "criteria": {
      "payments": "Checkout, billing, or payment processing issues.",
      "frontend": "Rendering, layout, or browser compatibility issues.",
      "account": "Login, permissions, or profile issues."
    }
  }
}
```

پاسخ Jev یک JSON کوچک است، نه یک پاراگراف (بعضی از فیلدها مثل id، model و usage برای کوتاه‌شدن خروجی در این نمونه حذف شده‌اند).

```
{
  "answers": {
    "is_bug": { "type": "noul", "noul": 0.96 },
    "team": {
      "type": "choice",
      "type": "choice",
      "confidence": 0.67,
      "probabilities": { "payments": 0.78, "frontend": 0.22, "account": 0 }
    }
  }
}
```

برای سؤال noul فقط یک عدد برمی‌گردد: احتمال درست بودن گزاره. برای choice و score، دو چیز متفاوت داریم: probabilities، توزیع احتمال مدل روی گزینه‌هاست و confidence -طبق مستندات OpenRouter- میزان متمرکز بودن همین توزیع را خلاصه می‌کند، نه درستی پاسخ را. در مثال بالا، بزرگ‌ترین احتمال ۷۸٪ است اما confidence برابر ۶۷٪ است؛ پس نباید فرض کرد این دو همیشه برابرند. همچنین confidence بالا لزوماً معنی امن بودن تصمیم را نمی‌دهد. از دید یک برنامه‌نویس، فراخوانی این API خیلی شبیه صدا زدن یک تابع معمولی است، نه یک مکالمه با یک چت‌بات. همان تیکت را می‌شود هم‌زمان با یک پرسش score هم پرسید، مثلاً «این تیکت چقدر فوری است؟» با سه سطح از «می‌تواند تا نسخه بعد صبر کند» تا «همین الان فروش را مسدود کرده است». همه در یک درخواست، بدون این‌که مدل مجبور باشد چیزی درباره‌ی چرایی تصمیمش توضیح دهد. همین الگو را می‌شود روی مسئله خودمان در بایت هم پیاده کرد: انتخاب خودکار هشتک برای یک مقاله، با یک choice که criteria اش هشتک‌های وب‌سایت بایت باشد.

می‌گویید ۸٪، حدود ۸۰٪ از این‌گونه موارد باید درست باشند. به همین دلیل می‌شود از عدد احتمال در منطق نرم‌افزار استفاده کرد؛ مثلاً با یک آستانه که بالای آن تصمیم خودکار اجرا شود و پابینش برای بازبینی انسانی بماند. اما دو تبصره مهم هستند: کالیبره بودن فقط برای داده‌هایی معتبر است که شبیه داده آموزش باشند و با تغییر توزیع داده می‌تواند خراب شود و آستانه را باید بر اساس هزینه تصمیم غلط در سیستم خودتان انتخاب کرده و روی داده خودتان بسنجید، نه یک عدد ثابت مثل ۹۰٪.

سرعت و هزینه

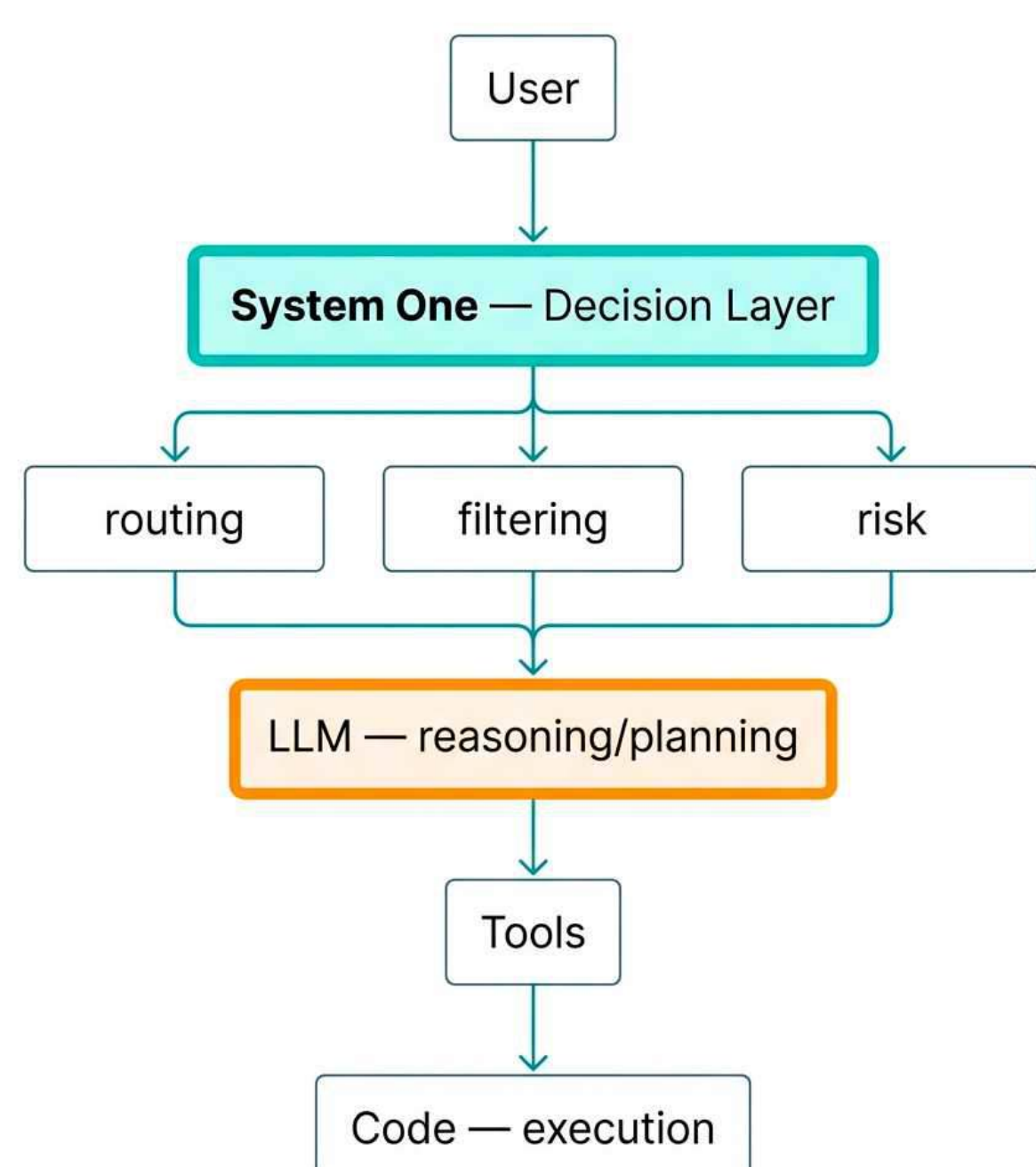
طبق پنج‌مارک‌های منتشرشده توسط TypeSafe:

- **تأخیر:** TypeSafe زمان پاسخ سرتاسر Jev را بین ۷۰ تا ۵۰۰ میلی‌ثانیه گزارش کرده، در برابر ۳ تا ۳۲۹ ثانیه برای LLM های frontier در این نوع وظایف. این دو بازه به مدل‌ها و وظایف متفاوتی برمی‌گردند و تقسیم مستقیم‌شان نسبت دقیق نمی‌دهد؛ خود TypeSafe سرعت‌افزایش را حدود ۴۰ تا ۲۰۰ برابر ذکر می‌کند و در یکی از workflow های ارزیابی‌اش ۱۹۳/۶ برابر سریع‌تر و ۴۴۴/۶ برابر ارزان‌تر را گزارش داده، که خودش آن را سقف دستاوردهای واقعی می‌داند. همه این اعداد ادعای خود شرکت است و تا این‌جا به‌صورت مستقل تأیید نشده‌اند.
- **نرخ خطای ساختاری:** چون فضای خروجی از پیش بسته است، پرسش «آیا خروجی از schema خارج شده؟» اصلاً معنا ندارد؛ TypeSafe این نرخ را صفر درصد گزارش کرده.

- **سقف context:** روی مدل Jev سقف context برابر با ۳۲'۰۰۰ توکن اعلام شده. یعنی برای ورودی‌های نسبتاً کوتاهی مثل یک تیکت یا یک مقاله کافی است، هر چند در مقایسه با پنجره context یک میلیون توکنی برخی LLM های امروزی، محدود است.

قیمت‌گذاری هم متناسب با همین طراحی است: در حال حاضر روی OpenRouter، هر یک میلیون توکن ورودی Jev حدود ۰/۴۲ دلار قیمت دارد و توکن خروجی رایگان است. چون خروجی دیگر متن بلند نیست، فقط چند مقدار عددی است. البته که این عدد را باید لحظه‌ای در نظر گرفت چون قیمت‌گذاری مدل‌های هوش مصنوعی معمولاً در طول زمان تغییر می‌کند.

احتمالاتی برای کل workflow می‌شود، نه فقط یک تابع برجسبزن. کد یا سیاستی که این احتمال‌ها را می‌خواند، می‌تواند تصمیم بگیرد کدام مسیر خودکار اجرا شود و کدام برای انسان نگه داشته شود. همین الگو در سطح یک ایجنت هم تکرار می‌شود. یک LLM برای هر انتخاب کوچک (کدام ابزار؟ کدام سند مرتبط است؟ این پیام مشکوک است یا نه؟) استدلال زبانی طولانی تولید نمی‌کند و این تصمیم‌ها به پرسش‌های typed تبدیل می‌شوند و لایه System One سریع پاسخ‌شان را می‌دهد؛ نتیجه دوباره به LLM یا کد برمی‌گردد تا مرحله بعد اجرا شود.



این تصویر کمک می‌کند یک برداشت رایج و نادرست تصحیح شود. System One قرار نیست یک LLM سریع‌تر باشد؛ یک جزء متفاوت در معماری یک ایجنت است. لایه‌ای که مسئولیتش تصمیم‌گیری است، نه تولید زبان یا اجرای نهایی. به همین دلیل، این نقش را می‌شود با چند اسم آشنا از دنیای یادگیری ماشین توصیف کرد که هر کدام همان الگوی رسیدن از state به انتخاب typed را دارند: verifier (آیا این خروجی معتبر است؟)، reward model (این پاسخ چقدر خوب است؟)، judge (کدام یک از این دو پاسخ بهتر است؟)، reranker (این سند چقدر به این پرسش مرتبط است؟) و router (این درخواست باید به کدام مسیر برود؟). نکته این است که همه این نقش‌ها را از قدیم با مدل‌های کوچک‌تر و اختصاصی هم می‌شد ساخت؛ حرف System One این است که یک API عمومی و از پیش آماده برای همه آن‌ها ارائه بدهد.

LLM + JSON Schema در Jev/Laya

پرسش طبیعی این است: «خب، یک LLM هم می‌تواند خروجی‌اش را با یک JSON Schema هماهنگ کند؛ پس تفاوت چیست؟»

فرق در جایی است که ساختار اعمال می‌شود. در ساده‌ترین و پرکاربردترین الگو، یعنی وقتی فقط در پرامپت از مدل می‌خواهیم خروجی را مطابق یک JSON Schema بدهد، مدل یک رشته آزاد تولید می‌کند و بعد آن رشته به یک قالب فشرده می‌شود.



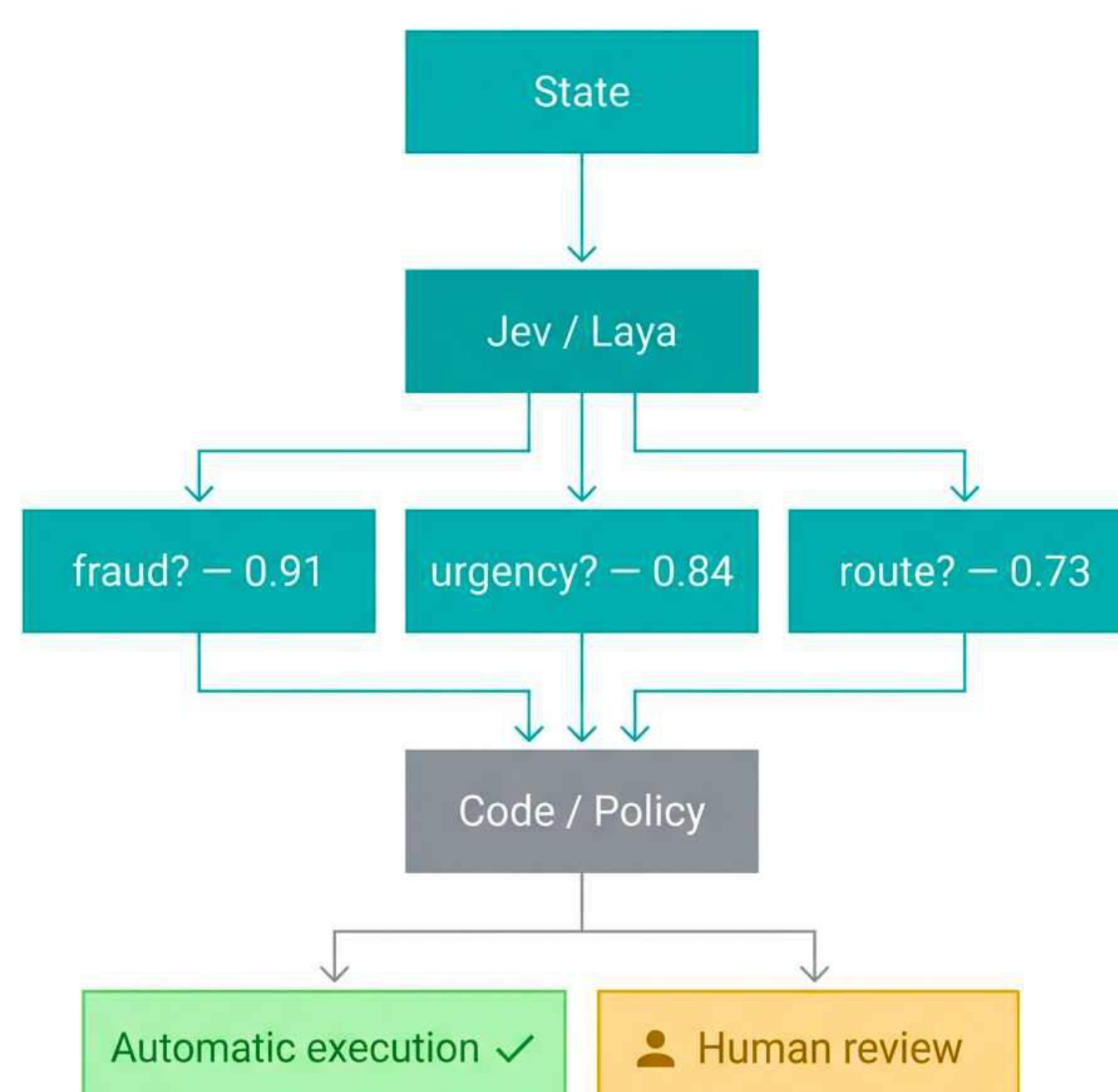
در System One، ساختار خروجی بخشی از خود مسئله مدل است، نه یک لایه بعدی.



TypeSafe این را strings در برابر typed values می‌نامد. فضای خروجی از پیش بسته است و مدل اصولاً نمی‌تواند خارج از type تعریف‌شده چیزی تولید کند. یک نقد منصفانه هم همین‌جا مطرح است: الگوی بالا تنها راه گرفتن خروجی ساختاریافته از LLM نیست. در constrained یا grammar-based decoding، مدل در هر مرحله فقط از میان توکن‌هایی نمونه‌برداری می‌کند که ساختار موردنظر (مثلاً JSON معتبر) را نمی‌شکنند؛ یعنی رشته از همان ابتدا مطابق ساختار تولید می‌شود، نه اینکه بعداً با parser اصلاح یا ردش کنیم. این روش هم چیز تازه‌ای نیست و سال‌هاست ابزارهایی برای آن وجود دارد. تفاوت واقعی System One، صرفاً محدود کردن نتیجه نهایی نیست؛ این که خود فرایند آموزش هم از ابتدا حول همان فضای محدود (و با یک هدف آموزشی متفاوت مثل RLCD) طراحی شده است. به همین دلیل، مقایسه درست‌تر، Jev/Laya در برابر یک LLM با decoding محدودشده است، نه فقط در برابر LLM + یک لایه اعتبارسنجی.

اما باید یک سوء تفاهم رایج را روشن کرد که type-safe بودن به معنی درست بودن نیست. اگر مدل تصمیم بگیرد یک تیکت به تیم «billing» تعلق دارد، از نظر type این پاسخ کاملاً معتبر است؛ اما خود تصمیم می‌تواند اشتباه باشد. ادعای صفر توهم^۱ که TypeSafe مطرح می‌کند هم دقیقاً همین معنا را دارد که مدل رشته آزاد و بی‌قاعده تولید نمی‌کند، نه این که هرگز تصمیم غلط نمی‌گیرد! خود شرکت هم این تمایز را در مستندات خود تصریح کرده است.

مدل‌های System One در برابر LLM + JSON Schema



انتخاب هشتگ یا مسیریابی یک تیکت، نمونه‌های ساده‌ای بودند: یک پرسش، یک پاسخ. اما طبق توضیح خود TypeSafe دربارهٔ بنچ‌مارک‌هایش، workflow های واقعی معمولاً از چند پرسش مستقل و تجزیه‌شده تشکیل می‌شوند که تصمیم نهایی، نه یک برجسب ساده، بلکه ترکیبی از چند احتمال است. این تصویر از یک دسته‌بند سریع بزرگ‌تر است. System One در این نقش یک لایه کنترل

این دو رویکرد لزوماً رقیب هم نیستند، بلکه می‌توانند در یک نرم‌افزار واحد کنار هم بنشینند. LLM برای فکر کردن و تولید زبان، System One برای تصمیم گرفتن سریع و کالبره‌شده و کد معمولی برای اجرا و کنترل.

بازیگران جدید؛ وقتی غول‌ها هم سکوت می‌کنند

خیلی زودتر از انتظار، ایده System One به استانداردی صنعتی در رابط‌های مدرن تبدیل شد. نگاهی به رده‌بندی زنده Decision Models در OpenRouter نشان می‌دهد که این فضا با سرعتی غافل‌گیرکننده در حال شلوغ شدن است. مهم‌ترین اتفاق در این جدول، ورود رسمی OpenAI با مدلی اختصاصی به نام GPT-6 Luna Decisions است. مدلی که به محض عرضه شدن، با ثبت بیش از ۶/۷ میلیون درخواست هفتگی در جایگاه دوم رقابت (پشت سر Jev با بیش از ۹۸۰ میلیون درخواست) نشست. حضور پرچم‌دار GPT در این دسته‌بندی حامل پیامی روشن است: حتی رهبران مدل‌های مولد هم پذیرفته‌اند که برای وظایفی مثل routing، امتیازدهی یا اعتبارسنجی سیاست‌ها، ائتلاف منابع برای تولید متن توجیه مهندسی و اقتصادی ندارد؛ بنابراین توسعه مدل‌هایی اختصاصی صرفاً برای خروجی‌های typed به یک ضرورت تبدیل شده است.

البته این میدان بازیگران متنوع دیگری هم پیدا کرده که هرکدام نیازی خاص را هدف گرفته‌اند:

کلادفلر با Clef و Clef Flash: با تمرکز بر تصمیم‌گیری فوق‌سریع و کم‌تأخیر در لبه شبکه.

لیکویید با d1: با تکیه بر معماری بهینه برای کانتکست‌های بلندتر و ارزیابی سیاست‌های حجیم.

پرپلکسیتی با سری Decider 27B در کنار گزینه‌های سبکی مثل Span-01 Lite و Kev 4B.

گرچه Jev به لطف پیش‌گامی‌اش هنوز بیش از ۹۷٪ سهم بازار را در دست دارد، اما پیام این صف‌آرایی روشن است: رقابت دیگر بر سر تولید متن‌های طولانی و انسانی نیست، بلکه بر سر این است که «کدام مدل با کم‌ترین تأخیر، کم‌ترین هزینه و بالاترین میزان کالبره بودن، یک تصمیم دقیق می‌گیرد».

۲. فضای تصمیم باید از قبل تعریف‌شده باشد و درست تعریف شده باشد. یک پرسش باز مثل «ایجنٹ باید چه کار کند؟» را نمی‌شود به‌طور مستقیم به این مدل‌ها داد؛ اول باید آن را به یک فضای محدود تبدیل کرد، مثلاً یک choice با گزینه‌های مشخص. تعریف این فضای تصمیم، همچنان کار مهندس سیستم است، نه مدل! اگر فضای تصمیم بد طراحی شود، نتیجه هم بی‌ارزش می‌شود. مثلاً برای پیام «می‌خواهم فردا ساعت پنج یک نفر از پشتیبانی با من تماس بگیرد»، یک پرسش noul مثل «آیا مشتری درخواست تماس با پشتیبانی انسانی دارد؟» جواب «بله» می‌دهد، اما همه اطلاعات مهم پیام مثل زمان دقیق تماس را دور می‌ریزد، چون اصلاً بخشی از فضای تصمیمی نبوده که تعریف کرده بودیم.

۳. تصمیم type-safe همچنان می‌تواند غلط باشد. همان‌طور که پیش‌تر اشاره شد، type-safe بودن به معنی correct بودن نیست.

۴. کالبره‌بودن جای دقت را نمی‌گیرد. یک مدل می‌تواند خیلی خوب کالبره باشد، یعنی وقتی می‌گوید ۷۰٪ مطمئن است، واقعاً حدود ۷۰٪ از این‌گونه موارد درست از آب دربیایند و در عین حال دقت متوسطی داشته باشد. این دو معیار مستقل‌اند: کالبره بودن درباره صداقت آماری اطمینان مدل است، دقت درباره درستی خود تصمیم. یک سیستم production باید هر دورا جداگانه اندازه بگیرد.

جمع‌بندی: بعد از عصر پر حرفی

Jev هنوز خیلی جوان است. خیلی زمان کمی از انتشار عمومی‌اش می‌گذرد و حتی خود اصطلاح System One هم تازه به این گستردگی وارد ادبیات AI شده. Laya راه دیگری را نشان می‌دهد: همان ایده، اما با معماری کاملاً مستند و قابل‌بازتولید. برای گفتن جمله «این نسل بعدی مدل‌های هوش مصنوعی است» هنوز زود است، اما ایده‌ای که این دو مطرح می‌کنند جدی است. در موج اول، هدف این بود که ماشین بتواند زبان انسان را بفهمد و تولید کند، رابطی از جنس متن به متن، اما آن‌چه اینجا می‌بینیم، رابط دیگری است: state به انتخاب احتمالاتی.

یک استفاده جالب‌تر که در یک بنچ‌مارک مستقل دیده شده، به‌کارگیری Jev به‌عنوان استخراج‌کننده ویژگی برای یک دسته‌بند جداگانه است، نه تصمیم‌گیر نهایی. این بنچ‌مارک را شخصی به نام anisselbd در ۱۷ سپتامبر ۲۰۲۶ روی ۲۰۰۰ ایمیل (نیمی فیشینگ، نیمی عادی) از مجموعه PhishNChips منتشر کرد. وقتی مستقیم پرسیده شد «آیا این ایمیل فیشینگ است؟»، Jev با دقت ۶۲٫۶٪ از Claude Haiku 4.5 با ۸۱٫۳٪ ضعیف‌تر بود. اما وقتی همان کار به پنج پرسش typed و باریک شکسته شد (ناهم‌خوانی دامنه، میزبانی رایگان، درخواست اطلاعات ورود، فوریت کاذب و فرستنده عمومی) و احتمال‌های خروجی وارد یک رگرسیون لجستیک ساده شدند که روی هزار ایمیل fit و روی هزار ایمیل جدا آزمایش شد، Jev به ۹۵٫۰٪ رسید. همین روش با Haiku به ۹۳٫۲٪ رسید و این تفاوت از نظر آماری معنی‌دار نبود ($p = ۰.۰۶۳$). پس نتیجه این نیست که Jev از Haiku بهتر است؛ نتیجه این است که شکستن یک پرسش مبهم به چند پرسش دقیق، Jev را از ضعیف به رقابتی رساند. چند محدودیت هم باید گفته شود: متن ایمیل‌ها مصنوعی بوده و سیگنال فیشینگ بیش‌تر در URL و فرستنده است؛ هر سیستم فقط با یک پرامپت آزمایش شده؛ و در همین آزمایش کالبره بودن، مقدار ECE برای Jev (برابر ۰٫۱۵۴) از Haiku (۰٫۰۹۷) بیش‌تر بود. در عوض، Jev ارزان‌تر و سریع‌تر بود (میانۀ تأخیر ۲۳۹ در برابر ۶۸۷ میلی‌ثانیه، و حدود ۰٫۳۸ در برابر ۰٫۴۶۲ دلار هزینه به‌ازای هر هزار ایمیل). این الگو نشان می‌دهد Jev و Laya لازم نیست همیشه «تصمیم نهایی» را بدهند؛ می‌توانند لایه میانی از سیگنال‌های عددی تولید کنند که مدل دیگری رویشان تصمیم نهایی را بگیرد.

اما System One برای همه چیز نیست

با همه این مزیت‌ها، این خانواده از مدل‌ها یک ابزار عمومی نیست. چند محدودیت مهم دارند:

۱. خروجی باز نمی‌تواند تولید کند. اگر مسئله چیزی مثل «این مقاله را در ۵۰۰ کلمه خلاصه کن» باشد، Jev ابزار مناسبی نیست. نه به این دلیل که ضعیف است، بلکه چون طراحی‌اش اصلاً برای این کار نیست. فضای خروجی این مدل همیشه از پیش بسته است.

میتونی مثل شاهنامه و با همون وزن چند بیت مثلاً در وصف جنگ پرتغال و سیب بسازی؟

حتماً! سعی می‌کنم حال و هوای شاهنامه و وزن متقارب معروفش را نگه دارم؛ یعنی نزدیک به «فعولن فعولن فعولن فعل»، با لحن حماسی ولی موضوع کاملاً مسخره!

رزم پرتقال و سیب

ز میدان برآمد خروش ستیز	ز سیب و ز پرتقال رزم خیز
چو سیبی به دندان برآراست تیغ	ز پرتقال آمد یلک بانگ میغ
بگفتا که «ای میوه سرخ روی	چه داری به جز برگ و بوی و گوی؟»
چنین گفت سیب: «ای ترش روی زرد	نبیخت دیگر در این دشت و گرد!»
برآشفت پرتقال و غرید سخت	که لرزید از بیم، شاخ درخت
ز یک سو سپاه سیب آمد پدید	ز دیگر سوی پرتقالان رسید
یلک سیب سرخ از کمین که برون	بزد بر پرتقال دشمن فسون
ولی پرتقال از نبردش ترس	بزد سیب را بر سر خویش، درس!
چنان جنگشان گشت سخت و دراز	که آتشوب افتاد در کوی و راز
سرانجام آمد یلک داور پیر	که گفتا: «بس است این نبرد دلیر!
شاه هر دو میوه، ز یک باغید	چرا بر سر هیچ در جنگید؟»
ز شرم آن دو جنگور نامدار	شدند آشتی و گشتند یار
کنون هر که بیند پرتقال و سیب	بگوید: «مخورشان، که دارند رقیب!»

بایت را ورق بزنید؛
این بار در فضای دیجیتال.

