

شماره نهم | شهریور ۱۴۰۵ | دانشگاه صنعتی شریف

byte Magazine



انجمن علمی دانشکده مهندسی کامپیوتر

نشریه علمی فرهنگی **بایت**

سرمقال؛

میان ماندن و رفتن

امیرحسین شهیدی

Bun.js؛ نـان تازه

اکوسیستم جاوااسکریپت!

معین آعلی

مترجمـان

نامرئـی

نیما شیرزادی

آیا سیلیکـون می تواند

رنـج بکشد؟

مهدی شاهمرادی

تاریخچهٔ معـماری

مجموعهٔ دستورالعمل ها

آرش یادگاری

پایان یک زامبی در سرور

وبسایت رویداد WSS

نیما شیرزادی

هیس! LLM ها

فریاد نـمی زنند.

آرش ماری اورباد

باگـز

بـازی

امیررضا جعفری

جـاسوس های

سیلیکـونی

نیما شیرزادی

خلاقیت از دل معماری: چگونه ساختار مدل های

تولید تصـویر، نوآوری را ممکن می کند؟

آروین بقال اصل

کالبد شکافـی

سیستم عامل در لحظه

نیما شیرزادی

هرچه بگندد نمکش می زنند،

وای به روزی که بگندد نمک

نیما شیرزادی

مدیر مسئول

امیرحسین شهیدی

سر دبیر

آرش شاه حسینی

صاحب امتیاز

انجمن علمی دانشکدهٔ مهندسی کامپیوتر دانشگاه صنعتی شریف

صفحه آرایـی و گرافیک

آرش شاه حسینی - امیررضا جعفری - نرگس کاری - فاطمه نیلفروشان - محمدپارسا آرانی - امیررضا اینانلو - معین آعلی

تخـریریه

آرمان طهماسبی - امیرحسین شهیدی - امیرحسین صوری - متین غیاثی - معین آعلی

نویسندگان

آرش یادگاری - نیما شیرزادی - معین آعلی - آروین بقال اصل - آرش ماری اورباد - مهدی شاهمرادی - امیررضا جعفری

ویراستاری

صهیب صادقی - امیرحسین صوری - آروین بقال اصل

فـنی

حسین زاهدی ادیب - معین آعلی



byte-mag.ir



byte-publication



byte-magazine



byte_mag

با ما همراه باشید

سرمقاله

میان ماندن و رفتن، ترس ایستاده بود...

امیرحسین شهیدی
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف
۳ دقیقه

این روزها که یکی یکی دوستانم چمدان می بندند و از این جا می روند، بیشتر از آن که به رفتن آن ها فکر کنم، به ماندن خودم فکر می کنم. به این که چه چیزی باعث شد آن ها با همه سختی ها، دلتنگی ها و ترس هایشان تصمیم بگیرند یک زندگی را پشت سر بگذارند و زندگی دیگری را شروع کنند و چه چیزی باعث شد من بمانم. اگر شرایطش را داشتم، شاید من هم مهاجرت می کردم. برای همین نمی توانم رفتن آن ها را صرفاً نتیجه متفاوت بودن اهداف شان بدانم. راستش را بخواهید، هرچه بیش تر فکر می کنم، کم تر تفاوتی میان هدف هایمان پیدا می کنم. همه مان آینده بهتری می خواستیم، رشد می خواستیم، زندگی می خواستیم و می خواستیم چیزی بیش تر از آن چه امروز داریم بسازیم.

پس تفاوت کجا بود؟

شاید در ارزش ها نبود. شاید در هدف ها هم نبود.

شاید تفاوت در چیزی بود که کمتر درباره اش حرف می زنیم: ترس.

من ترسیدم.

از تنهایی. از دوری. از جدا کردن تکه هایی از قلبم که بیش از ۲۰ سال برای کنار هم چیدن شان زحمت کشیده بودم. شاید حتی از این که در جایی تازه شکست بخورم و هیچ کس آن جا مرا نشناسد و در روزهای سختم مرا به آغوش بکشد.

من ترسیدم؛ از این که بروم و دیگر نتوانم سهمی در تغییر چیزی داشته باشم. از این که یک روز از پشت صفحه ای کوچک، خبر خراب شدن کشورم را در گوشه ای دیگر از دنیا ببینم.

دقیق نمی دانم از چه می ترسیدم؛ اما می دانم ترسیدم. حتی از معنای خود رفتن هم ترسیدم؛ از این که مهاجرت برای من شبیه اعتراف باشد: این که شاید دیگر امیدی به درست شدن کشورم ندارم.

ما معمولاً دوست نداریم ترس را در تصمیم هایمان ببینیم. ترجیح می دهیم اسمش را «منطق»، «شرایط»، «مسئولیت» یا «واقع بینی» بگذاریم. اما گاهی لازم است صادق باشیم و بگوییم: آن کار را نکردم، چون ترسیدم!

این اعتراف لزوماً شکست نیست. شاید اولین قدم خودشناسی باشد. چون پیش از آن که بدانیم کجا باید برویم، باید بدانیم برای چه چیزی حاضریم سختی بکشیم. گاهی تفاوت آدم ها در چیزی که می خواهند نیست؛ در چیزی است که حاضرند برای خواسته شان از آن بگذرند.

ارزش های ما فقط چیزهایی نیستند که درباره شان حرف می زنیم؛ ارزش واقعی هر چیز را، هزینه ای که حاضر به پرداختنش هستیم مشخص می کند؛ حتی علم نیز از همین قاعده مستثنی نیست. علم ارزشمند و مقدس است، اما ارزش علم فقط در دانستن نیست. مهم این است که بدانیم دانش مان را در خدمت چه چیزی قرار می دهیم.

اگر قرار است بهترین سال های زندگی مان را صرف یادگیری و ساختن کنیم، باید بدانیم چیزی که می سازیم، با آن چه به آن باور داریم چه نسبتی دارد. شاید بزرگ ترین اشتباه این باشد که سال ها برای رسیدن به هدفی بجنگیم که حتی نمی دانیم

چرا انتخابش کرده ایم.

بدترین گم شدن این نیست که راه را پیدا نکنیم؛ این است که به مقصدی برسیم که هیچ وقت نمی خواستیم! اگر از مسیر خسته ایم، شاید لازم نیست همیشه بیش تر تلاش کنیم. شاید باید بایستیم و از خودمان بپرسیم: آیا دارم برای چیز درستی می جنگم؟

هیچ کس نمی تواند نسخه ای یکسان برای همه بپیچد. یکی باید برود، دیگری باید بماند. یکی باید بسازد، دیگری باید مسیرش را عوض کند. مسئله این نیست که کدام انتخاب از نظر دیگران درست تر است؛ مسئله این است که کدام انتخاب با خود واقعی ما صادق تر است. شاید خوشبختی هم همین باشد؛ نه رسیدن به نقطه ای که دیگر هیچ مشکل و درد و ترسی نداریم، بلکه قرار گرفتن در مسیری که سختی هایش را آگاهانه انتخاب کرده ایم.

و شجاعت به معنی نبودن ترس نیست؛ شجاعت یعنی ترسیدن و باز هم قدم برداشتن. شاید زندگی همین باشد: پیدا کردن چیزی که ارزش ترسیدن را داشته باشد. چیزی که آن قدر برایمان مهم باشد که با وجود ترس، باز هم انتخابش کنیم چون گاهی قرار نیست ترس را از بین ببریم؛ باید یاد بگیریم از میانش عبور کنیم. به قول فرانک هربرت:

«من نباید بترسم. ترس، قاتل ذهن است. ترس همان مرگ کوچکی است که نابودی کامل را به همراه می آورد. با ترسم روبه رو خواهم شد. اجازه می دهم از من عبور کند و از میان وجودم بگذرد. و هنگامی که از من گذشت، چشم درونم را می گشایم تا مسیرش را ببینم. آن جا که ترس رفته است، دیگر چیزی باقی نخواهد ماند؛ تنها من خواهم ماند.»



نیما از بایت به شدت ناراحته
چون توی اسال اخیر ۵ تا متن داده و ما
یادمون رفته منتشر کنیم

ما هم تصمیم گرفتیم این شماره رو
مخصوص اقا نیما بزنیم تا از دلش در بیاد



تاریخچه معماری مجموعه دستورالعمل‌ها^۱

آرش یادگاری
دانشگاه صنعتی شریف
کارشناسی ۱۳۹۹

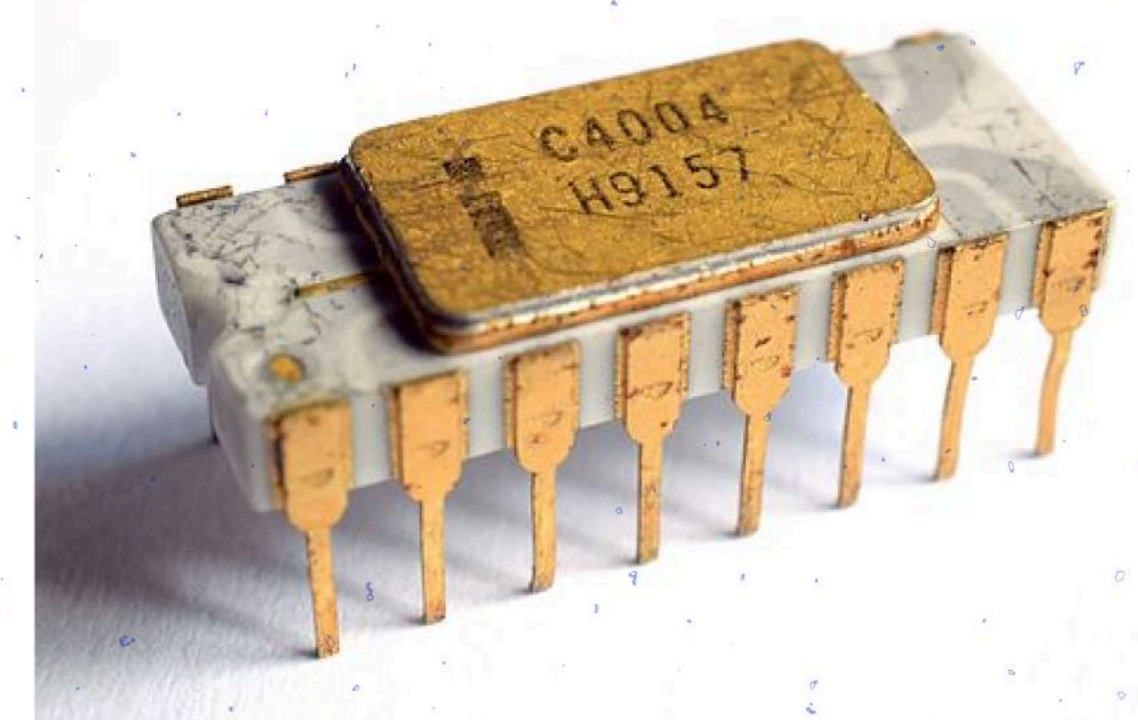


شاید در درس‌هایی مثل «ساختار کامپیوتر» یا «معماری کامپیوتر» با ISA آشنا شده باشید یا حتی نمونه‌ای ساده از آن را پیاده‌سازی کرده باشید. ISA ها استانداردهایی هستند و بیانگر مجموعه دستورالعمل‌هایی هستند که یک پردازنده می‌تواند انجام دهد. اما معماران کامپیوتر چگونه ISA ها را طراحی می‌کنند و یا تکامل می‌دهند؟ نخستین ISA ها چگونه شکل گرفتند؟ و امروزه دستگاه‌های دیجیتال پیرامون ما بر پایه کدامیک از آن‌ها ساخته می‌شوند؟ در این متن می‌کوشیم به این پرسش‌ها پاسخ دهیم.

سفرمان را به دهه‌های ۶۰ و ۷۰ میلادی آغاز می‌کنیم. زمانی که غول‌هایی مانند IBM بازار کامپیوتر را در دست داشتند. آن روزها اکوسیستم نرم‌افزار مانند امروز بالغ نشده بود و هر برنامه معمولاً برای سخت‌افزار مشخصی نوشته می‌شد؛ بنابراین تعویض سیستم اغلب به معنای بازنویسی برنامه‌ها بود.

در سال ۱۹۶۴، IBM با معرفی System/360 بازی را عوض کرد: یک معماری واحد برای خانواده‌ای از سیستم‌ها؛ یعنی مشتریان می‌توانستند نرم‌افزارهایشان را بدون بازنویسی روی مدل‌های مختلف اجرا کنند. چند سال بعد، Intel 4004 نخستین ریزپردازنده ۴-بیتی را روی یک تراشه عرضه کرد و در ادامه خانواده ۸۰۸۶ (یا ۸۰۸۶ در سال ۱۹۷۸) به معماری غالب رایانه‌های شخصی بدل شد.

وجود استانداردهایی مانند ISA سبب شد توسعه‌دهندگان انواع نرم‌افزارها از سیستم‌عامل گرفته تا برنامه‌های تحت وب، بدون آن‌که درگیر جزئیات پیاده‌سازی هر



پردازنده شوند، تنها با تکیه بر مجموعه دستورالعمل‌های پشتیبانی‌شده، نرم‌افزار بنویسند.

برای طراحی یک ISA باید هم‌زمان به نیازهای سخت‌افزاری و نرم‌افزاری توجه کرد. معماری موفق باید روی طیف وسیعی از پردازنده‌ها با اهداف گوناگون (از مصرف کم تا کارایی بالا) قابل پیاده‌سازی باشد و در عین حال، واسط مناسبی برای کامپایلرها و مترجم‌ها فراهم کند تا برنامه‌ها را به دستورالعمل‌های مناسب تبدیل کنند. همین نیازهای متنوع — و گاه متضاد — باعث شده‌است رویکردهای مختلفی در طراحی ISA شکل بگیرند.

دهه ۸۰ میلادی آغاز یکی از جدل‌های ماندگار بود: RISC^۲ در برابر CISC^۳. گروهی از مهندسان IBM نشان دادند بسیاری از برنامه‌ها عمدتاً از تعداد محدودی دستور ساده استفاده می‌کنند؛ پس با ساده‌سازی ISA و بهینه کردن مسیر اجرا، می‌توان به کارایی بالاتری رسید.

این نگاه به شکل‌گیری خانواده‌هایی چون MIPS -دستورالعمل استفاده‌شده در کنسول بازی PlayStation 2- انجامید و بعدها ARM با تکیه بر همین سادگی و بهره‌وری انرژی، انتخاب غالب تلفن‌های همراه شد. RISC-V از دیگر مجموعه دستورالعمل‌های رایج این دسته است که به‌صورت متن‌باز عرضه می‌شود. این بدان معناست که می‌توان پردازنده‌های مبتنی بر این ISA را بدون خرید مجوز، طراحی و تولید کرد.

ISA ها نیز با توجه به نیازهای جدید برنامه‌ها تکامل می‌یابند. این تکامل معمولاً به‌صورت افزونه‌هایی روی ISA پایه ارائه می‌شود. نمونه شاخص آن، پردازش برداری یا SIMD^۴ است: «یک دستور برای چندین داده»؛ رویکردی که برای شتاب دادن به کارهای گرافیکی، پردازش سیگنال، رمزنگاری و یادگیری ماشین به کار می‌رود.

در دنیای x86، این نقش را خانواده SSE و سپس AVX/AVX2/AVX-512 بر عهده دارند (روی پردازنده‌های اینتل و AMD). در ARM ابتدا NEON معرفی شد و در نسل‌های جدیدتر، SVE/SVE2 با طول بردار انعطاف‌پذیر، همین مسیر را ادامه داده‌اند. در RISC-V نیز افزونه V قابلیت‌های برداری را به‌صورت ماژولار و با طول بردار پویا ارائه می‌کند. کامپایلرها با بررسی پشتیبانی از انواع

افزونه‌ها، از دستورات این افزونه‌ها برای افزایش سرعت اجرای برنامه استفاده می‌کنند. شما می‌توانید وجود این افزونه را با دستور lscpu و مشاهده قسمت flags بررسی کنید.

```
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 sht tm pbe syscall nx pdpe1gb rdtscp lm constant tsc art arch perfmon pbs bts rep good nopl xtopology nops top tsc cpl uid aperfmperf pni pclmulqdq dtes64 monitor ds cpl vmx est tm2 sse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc deadline timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd ibrs ibpb stibp ibrs enhanced tpr sh adow flexpriority ept vpid ept ad fsgsbase tsc adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify hwp_act_window hwp_epp vnmi md clear flush lld arch capabilities
```

انتخاب ISA برای طراحی یک پردازنده، تابع مجموعه‌ای از ملاحظات فنی و تجاری است. بلوغ و وسعت اکوسیستم نرم‌افزاری و سازگاری با برنامه‌های موجود، مدل مجوز و هزینه بهره‌برداری (یا متن‌باز بودن) استاندارد، پیچیدگی و هزینه پیاده‌سازی و اهدافی نظیر کارایی/مصرف توان و عوامل اجرایی مانند زمان عرضه، برخی از عوامل مؤثر در این زمینه هستند.

به‌طور مثال، در سال‌های اخیر، سرورهای مبتنی بر RISC-V به دلیل متن‌باز بودن و وجود افزونه‌ها با دستوراتی برای رمزنگاری^۵ و محاسبات برداری مورد استقبال قرار گرفته‌اند.

سفر ISA از سیستم‌های یکپارچه و انحصاری تا دنیای متنوع و رقابتی امروز، داستان تلاشی بی‌وقفه برای یافتن بهترین زبان مشترک میان نرم‌افزار و سخت‌افزار است.

روایتی که همچنان ادامه دارد و نشان‌گر تحول دنیای کامپیوترهاست.

نیما در حال آتش زدن یه اتاق
چون معتقده بازدهی کافی رو ندارن



کردن دقیق گلوگاه، بدون این که حتی یک خط کد دیباگ در برنامه تان نوشته باشید!

روی تئوری PMCA: چالش های اعصاب خردکن

البته در دنیای عملی همه چیز به این قشنگی نیست و درگیر شدن با PMC ها در محیط پروداکشن چالش های خاص خودش را دارد. فقر منابع فیزیکی: از آن جا که این شمارنده ها قطعات فیزیکی، ترانزیستورهای واقعی و گران قیمت روی سطح سیلیکونی CPU هستند، تعدادشان به شدت محدود است. شما نمی توانید همزمان ۵۰ متریک مختلف را مانیتور کنید و در نهایت باید به همان ۴ تا ۸ رجیستری که روی هر هسته قرار دارد، قناعت کنید.

تکنیک مالتی پلکسینگ^۲ و افت دقت: اگر بخواهید رویدادهای بیشتری را همزمان مانیتور کنید (مثلاً ۱۵ متریک با ۴ رجیستر)، سیستم عامل مجبور است همواره این رجیسترهای محدود را بین متریک های مختلف جابه جا کند. اصطلاحاً به این کار Time-slicing می گویند. این جابه جایی های مداوم باعث می شوند بخشی از رویدادها را در زمان خاموش بودن متریک از دست بدهید. در این حالت، لینوکس مجبور است اعداد را تخمین بزند (Scaling) که این باعث افت شدید دقت داده خروجی می شود.

جهنم وابستگی به معماری: همان طور که از کلمه «Model-Specific» در نام MSR پیداست، این رجیسترها و کدهای رویداد آن ها به شدت وابسته به مدل دقیق پردازنده هستند. یعنی کدی که در پردازنده Intel Skylake برای مانیتور کردن پرش های اشتباه می زنید، در پردازنده AMD Zen 3 یا حتی در نسل جدیدتر خود اینتل ممکن است کاملاً متفاوت باشد. این نبود استاندارد باعث می شود ابزارها و اسکریپت های مانیتورینگ شما پرتابل نباشند و با جابه جایی سرور، به راحتی از کار بیفتند.

به طور خلاصه، ابزارهایی مثل perf stat بخش زیادی از این چالش های سخت افزاری را برای ما پنهان می کنند و یک خروجی تمیز تحویل می دهند؛ اما پشت پرده، این نبرد ثانیه ای رجیسترهاست که قدرت واقعی سیستم را نمایان می کند.

• هر زمان، پیش بینی اجرای یک حلقه اشتباه از آب درآمد، این جا ثبت کن.
• هر زمان، پردازنده منتظر رسیدن دیتا از RAM ماند و اصطلاحاً بلاک شد، کلاک های هدررفته را بشمار.

از این لحظه به بعد، سیستم عامل کنار می کشد و خود سخت افزار CPU به صورت فیزیکی، شروع به شمردن این رویدادها می کند. لینوکس فقط گاهی (مثلاً هر یک ثانیه)، عدد داخل این رجیسترها را می خواند و به شما نمایش می دهد. این، یعنی شما بدون هیچ سربار نرم افزاری، در حال مانیتور کردن عمیق ترین اتفاقات داخل پردازنده هستید؛ قابلیتی که در بهینه سازی کدهای Rust، C++ یا پایگاه های داده عظیم واقعاً یک معجزه است.

ابزار perf: پنجره ای به درون پردازنده

برای این که بتوانیم از این جادوی سخت افزاری استفاده کنیم، لینوکس ابزار قدرتمندی به نام perf را در اختیار ما قرار داده است. این ابزار دقیقاً همان مترجمی است که خواسته های ما را به رجیسترهای MSR می فهماند و استفاده از آن معمولاً در دو فاز اصلی انجام می شود.

دیدن نمای کلی با perf stat: اگر می خواهید بدانید برنامه شما در حالت کلی چقدر منابع پردازنده را هدر داده است، کافی است دستور `perf stat -d ./my_program` را در ترمینال اجرا کنید. به محض اجرای برنامه، گزارشی می بینید که نشان می دهد کد شما چند میلیون بار در خواندن Cache شکست خورده، چند دوره پردازشی را طی کرده و چه تعداد پرش اشتباه داشته است.

نقطه زنی با perf record و perf report: اگر perf stat به شما بگوید که cache-miss بالایی دارید، پرسش بعدی این است: «دقیقاً کجای کد من این مشکل را ایجاد کرده است؟» این جاست که دستور `perf record` وارد می شود. این دستور در حین اجرای برنامه، هزاران بار در ثانیه از وضعیت پردازنده عکس برداری می کند، سپس با اجرای `perf report` می توانید دقیقاً به خطی (یا تابعی) از کد برسید که بیشترین زمان پردازنده یا بیشترین cache-miss را به خود اختصاص داده است؛ یعنی پیدا

جاسوس های سیلیکونی

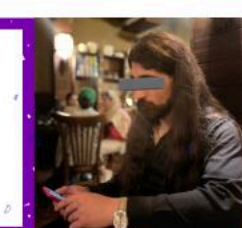
پردازنده ها چگونه خودشان را مانیتور می کنند؟

دانشگاه صنعتی شریف

نیما شیرزادی

۴ دقیقه

کارشناسی ارشد ۱۴۰۱



جادوی PMC ها و نفوذ به عمیق ترین لایه های CPU

شاید تا به حال برایتان این پرسش پیش آمده باشد که سیستم عامل اصلاً از کجا می فهمد پردازنده چند بار در خواندن دیتا از Cache، اصطلاحاً Hit داشته یا Miss؟ چند بار پرش اشتباه^۲ انجام داده؟ چقدر در اجرای دستورات بی کار مانده و منابعش قفل شده است؟ یا شاخص حیاتی IPC^۳ (تعداد دستورالعمل های اجرا شده در هر دوره کلاک) را چطور با این دقت محاسبه می کند؟

شاید فکر کنید لینوکس یا ویندوز با اجرای کد نرم افزاری این آمار را می گیرند؛ اما در این صورت، فاجعه رخ می داد. همین «سربار شمردن» باعث می شد پردازنده تمام وقتش را صرف شمردن کارهای خودش کند و منابع سیستم فلج شوند! در مهندسی نرم افزار به این پدیده «اثر ناظر»^۴ می گویند؛ یعنی خود مانیتورینگ، رفتار سیستم را تغییر می دهد.

برای حل این بن بست، مهندسان سخت افزار یک شاهکار تکنولوژی به نام PMC را وارد معماری پردازنده ها کردند که دقیقاً نقش یک دستگاه مانیتورینگ بدون سربار را برای مغز سیستم بازی می کند.

ماجرا از این قرار است که غول هایی مثل اینتل، AMD و ARM در همان لایه سیلیکونی و فیزیکی CPU، یک تعداد رجیستر سخت افزاری بسیار خاص تعبیه کرده اند که به آن ها MSR^۵ می گویند. وقتی شما ابزار قدرتمندی مثل perf را در لینوکس اجرا می کنید، سیستم عامل به طور مستقیم با سخت افزار حرف می زند و به یکی از این رجیسترهای کنترلی یک «کد رویداد»^۶ هگزادسیمال می دهد. سیستم عامل به زبان سخت افزار می گوید:

از الان، هر زمان دیتایی در L3 Cache پیدا نشد، یک واحد به این رجیستر اضافه کن.

Bun.js؛ نان تازه اکوسیستم جاوااسکریپت!

معرفی یک Runtime دیگر جاوااسکریپت



دانشگاه صنعتی شریف

معین آعلی

۱۳ دقیقه

کارشناسی ۱۴۰۱

اگر مدتی در دنیای JavaScript برنامه نویسی کرده باشید، احتمالاً میز کارتان با ابزارهای مختلفی شلوغ شده است. یک Runtime برای اجرای برنامه، یک Package Manager برای نصب وابستگی‌ها، ابزاری برای تست، ابزاری برای Bundle کردن و شاید چند ابزار دیگر برای توسعه و انتشار پروژه. این تنوع یکی از قدرت‌های اکوسیستم JavaScript است؛ اما گاهی همین قدرت، پیچیدگی ایجاد می‌کند. برای راه‌اندازی یک پروژه نسبتاً ساده ممکن است مجبور شویم چند ابزار مستقل را نصب، تنظیم، و با یکدیگر هماهنگ کنیم. حالا یک نان تازه از تنوع اکوسیستم JavaScript بیرون آمده که می‌خواهد بخش زیادی از این ابزارها را در یک بسته قرار دهد.

Bun یک JavaScript Runtime و Toolkit همه‌کاره است که Runtime، پکیج‌منیجر، Test Runner و Bundler را در یک فایل اجرایی مستقل جمع می‌کند. ایده اصلی آن را می‌توان در سه کلمه خلاصه کرد: سرعت، یکپارچگی و سازگاری. اما آیا Bun واقعاً آن قدر سریع است که ادعا می‌کند؟ آیا در پروژه‌های واقعی هم استفاده می‌شود؟ و مهم‌تر از همه، آیا این نان تازه می‌تواند جایی در سفره‌ای پیدا کند که سال‌هاست Node.js بر آن حکومت می‌کند؟

Bun چیست؟

برای شناخت Bun ابتدا باید مفهوم JavaScript Runtime را بشناسیم. مرورگرهایی مانند Chrome و Firefox محیطی برای اجرای JavaScript فراهم می‌کنند؛ اما برای اجرای JavaScript خارج از مرورگر، مثلاً روی یک سرور، به محیط دیگری نیاز داریم. مشهورترین پاسخ این مسئله در پانزده سال گذشته Node.js بوده است. Node.js امکان اجرای JavaScript در خارج از مرورگر را فراهم کرد

و به مرور اکوسیستم عظیمی از ابزارها و کتابخانه‌ها پیرامون آن شکل گرفت. Bun نیز یک JavaScript Runtime است، اما نمی‌خواهد فقط جایگزینی برای اجرای کد باشد. در یک فایل اجرایی، چهار ابزار اصلی را در اختیار توسعه‌دهنده می‌گذارد:

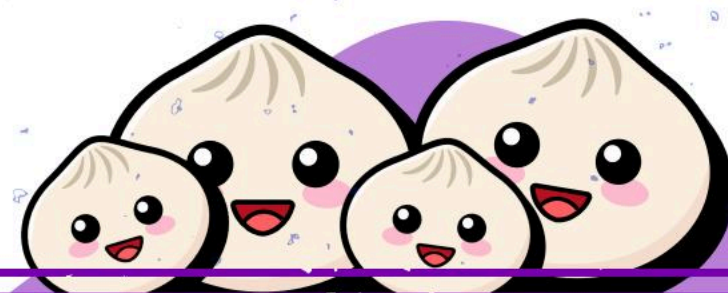
۱. Runtime برای اجرای JavaScript و TypeScript.

۲. Package Manager به عنوان جایگزینی برای npm و yarn و غیره.

۳. Test Runner به عنوان جایگزینی برای ابزارهایی مانند Jest و Vitest.

۴. Bundler برای کاری که معمولاً ابزارهایی مانند esbuild، Vite یا Webpack انجام می‌دهند.

نکته مهم این است که برای استفاده از Bun مجبور نیستیم همه این ابزارها را یک‌باره بپذیریم. برای مثال، می‌توان در یک پروژه Node.js همچنان Node.js Runtime نگه داشت، اما از bun install برای نصب پکیج‌ها یا از bun test برای اجرای تست‌ها استفاده کرد. به بیان دیگر، Bun نمی‌گوید میز کار قدیمی خود را دور بیندازید؛ پیشنهاد می‌کند کم‌کم چند وسیله روی آن را با یک ابزار واحد جایگزین کنید.



داخل این نان چه خبر است؟

یکی از تفاوت‌های بنیادی Bun و Node.js به موتور JavaScript آن‌ها بازمی‌گردد. Node.js از V8، موتور JavaScript توسعه‌یافته توسط Google و مورد استفاده در Chrome، بهره می‌برد. Bun به جای V8 از JavaScriptCore استفاده می‌کند؛ موتوری که بخشی از پروژه WebKit است و Safari نیز از آن استفاده می‌کند. خود Bun نیز عمدتاً با زبان Zig توسعه یافته است (البته به تازگی آن را با Rust بازنویسی کرده‌اند)؛ یک زبان برنامه‌نویسی سطح پایین که امکان کنترل دقیق روی حافظه و عملکرد را در اختیار توسعه‌دهنده قرار می‌دهد. اما شاید مهم‌تر از انتخاب بین JavaScriptCore یا Zig، فلسفه طراحی Bun باشد. در اکوسیستم سنتی JavaScript بسیاری از ابزارها مستقل از یکدیگر ساخته شده‌اند؛

Package Manager یک پروژه است، Runtime پروژه‌ای دیگر و Test Runner Bundler نیز مسیر خودشان را دارند. در Bun، این اجزاء از ابتدا برای کار کردن در کنار یکدیگر طراحی شده‌اند. برای مثال، می‌توان یک فایل TypeScript را به طور مستقیم با دستور `bun run server.ts` اجرا کرد؛ بدون این‌که برای چنین استفاده‌ای به مرحله Build جداگانه یا ابزاری مانند `ts-node` احتیاج داشته باشیم.

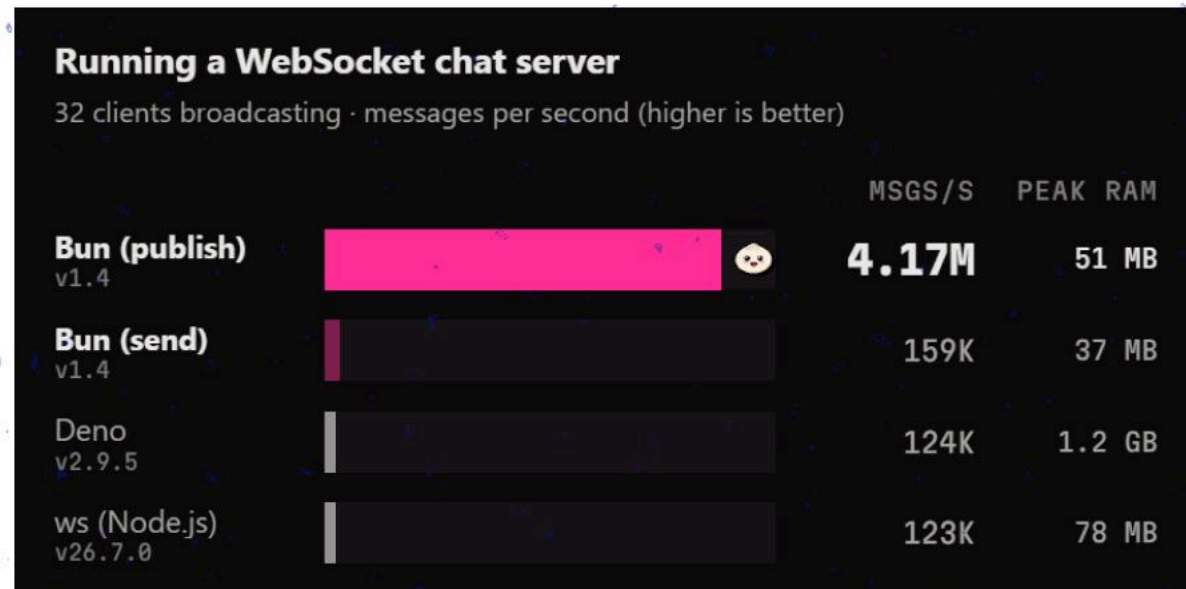
Bun در عین حال، تلاش می‌کند با دنیای Node.js سازگار بماند. از `node_modules` و پکیج‌های npm پشتیبانی می‌کند و بسیاری از API های Node.js را پیاده‌سازی کرده است. این موضوع اهمیت زیادی دارد، زیرا ارزش یک Runtime فقط به سرعت اجرای JavaScript نیست؛ کتابخانه‌ها و ابزارهایی که می‌توان روی آن اجرا کرد نیز بخشی از ارزش آن هستند.

یک نان با چند طعم؛ جعبه ابزار Bun

Package Manager شاید یکی از اولین بخش‌هایی باشد که توسعه‌دهندگان بدون مهاجرت کامل به Bun امتحان می‌کنند. `bun install` همان کاری را انجام می‌دهد که معمولاً از `npm install`، `yarn` یا `pnpm install` انتظار داریم و همچنان از اکوسیستم npm استفاده می‌کند. Bun یک Test Runner داخلی نیز دارد. دستور `bun test` برای توسعه‌دهندگانی که با Jest کار کرده‌اند ناآشنا نیست؛ قابلیت‌هایی مانند Mock، `expect()`، Snapshot، تست DOM و Coverage در آن وجود دارند. در کنار آن، Bundler داخلی Bun می‌تواند TypeScript، JSX، CSS و HTML را برای مرورگر و سرور Bundle کند. حتی امکان Compile کردن برنامه به یک فایل اجرایی مستقل وجود دارد؛ به این معنی که Runtime موردنیاز برنامه نیز همراه آن فایل توزیع می‌شود.

قابلیت‌های داخلی Bun از این هم فراتر رفته‌اند. `Bun.serve()` برای HTTP و WebSocket، کلاینت داخلی SQL برای MySQL، PostgreSQL و SQLite، کلاینت Redis، دسترسی به فضای ذخیره‌سازی سازگار با S3، ابزار Shell و حتی قابلیت‌های Hashing برای رمز عبور

اما وقتی از قابلیت داخلی Pub/Sub و publish در Bun استفاده شده، عدد به حدود ۴/۱۷ میلیون پیام در ثانیه رسیده است. در مصرف حافظه نیز Bun برای حالت publish حدود ۵۱MB گزارش شده است.



البته عدد ۴/۱۷ میلیون را نباید بدون توضیح کنار ۱۲۳ هزار قرار داد و نتیجه گرفت «Bun دقیقاً ۳۴ برابر از Node سریع تر است». publish از مسیر بهینه شده داخلی Pub/Sub در Bun استفاده می کند و معماری آن با اجرای یک حلقه send یکسان نیست.

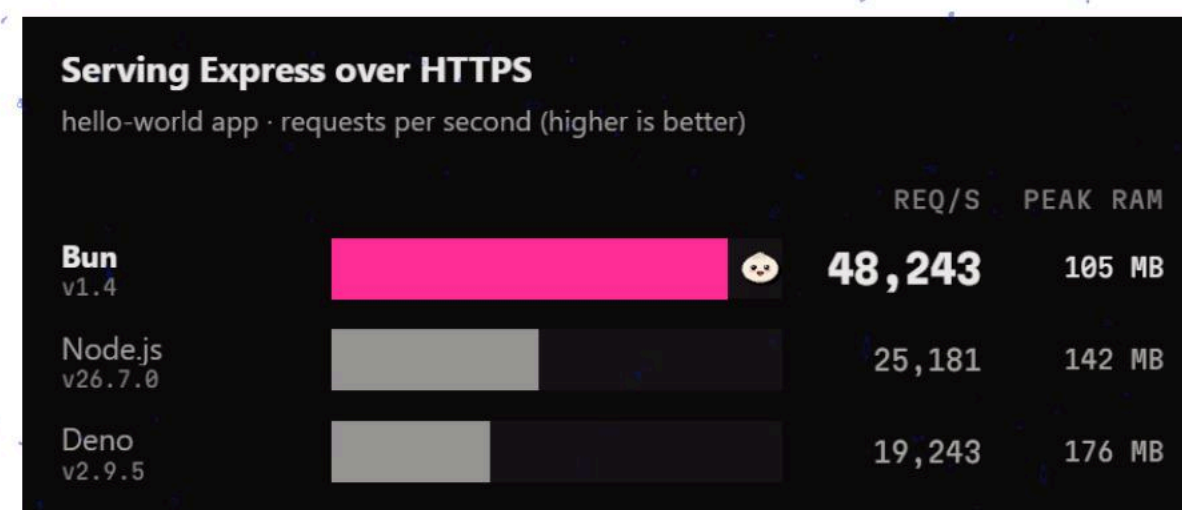
از تور تا Production؛ چه کسانی Bun را واقعاً استفاده می کنند؟

شاید این بخش از داستان Bun از نمودارهای Performance هم مهم تر باشد. فناوری وقتی جدی می شود که از بنچمارک خارج شده و وارد Production شود. صفحه رسمی Bun سه نمونه برجسته را معرفی می کند: Railway، Midjourney، Claude Code به صورت یک Single-file Executable مبتنی بر Bun توزیع می شود. Bun Runtime داخل همان فایل قرار می گیرد؛ بنابراین کاربر برنامه را دریافت و اجرا می کند و نیازی به نصب جداگانه Runtime ندارد. نمونه دوم، Midjourney، تمام نوتیفیکیشن های تصاویر Midjourney از WebSocket Server مبتنی بر Bun عبور می کنند. قابلیت های Pub/Sub و Backpressure و فشرده سازی پیام نیز مستقیماً در Bun.serve() قرار دارند و در این معماری نیازی به ws یا Socket.IO نیست. نمونه سوم، Railway، است که محصول Serverless Functions خود را بر پایه Bun ساخته است. یک Railway Function می تواند به طور مستقیم یک فایل TypeScript باشد؛ Bun آن را بدون مرحله Build جداگانه اجرا می کند و این موضوع برای محیط Serverless، که زمان Cold Start اهمیت زیادی دارد، جذاب است.

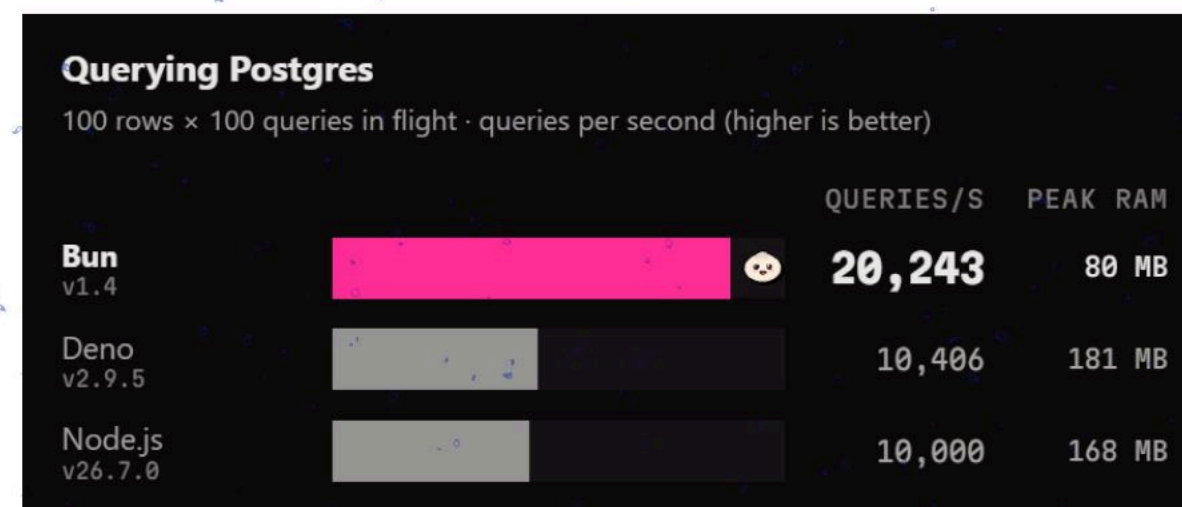
پردازنده AMD EPYC 9R14 انجام داده است.

برش دوم بنچمارک؛ HTTP و پایگاه داده

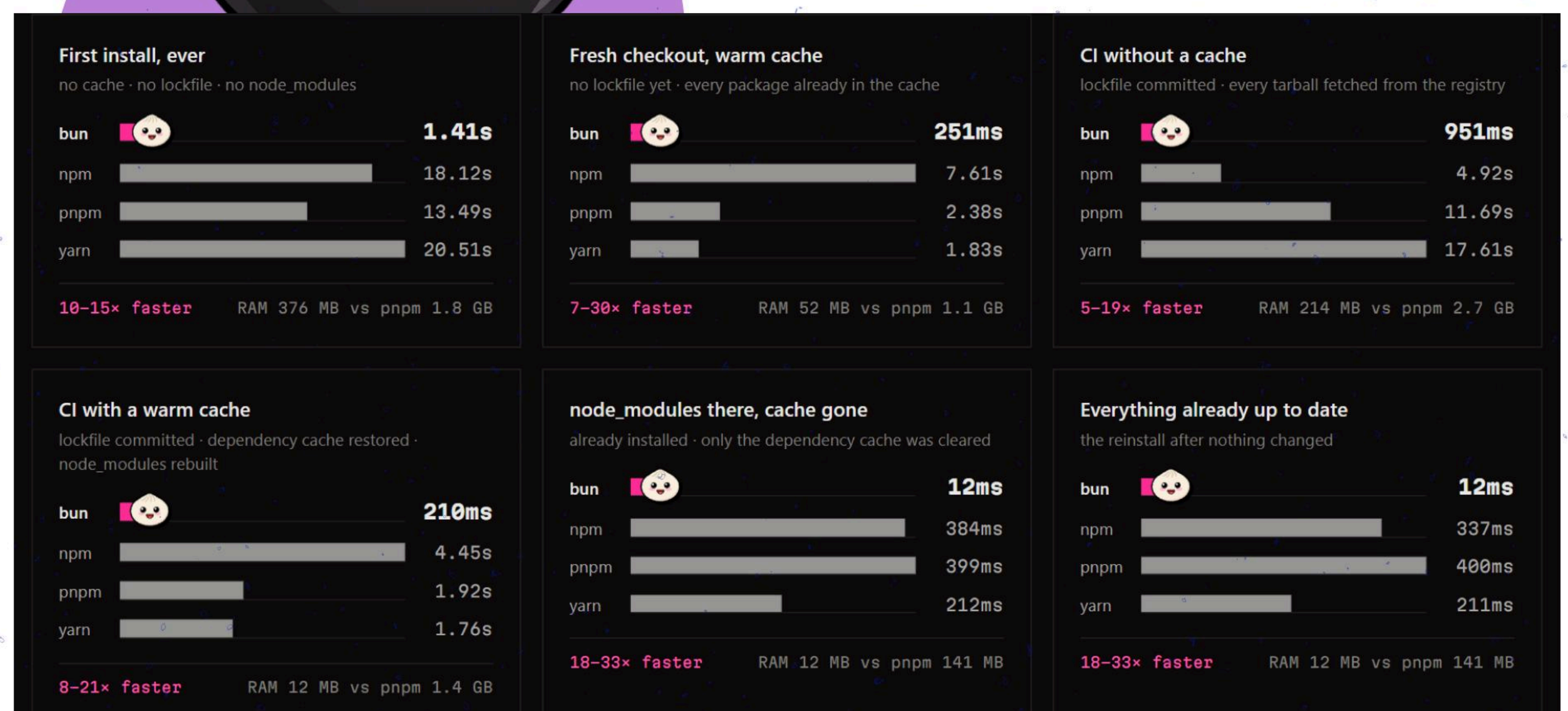
سرعت Package Manager فقط بخشی از ماجراست. Bun عملکرد Runtime را نیز با Node.js و Deno مقایسه کرده است. در بنچمارک یک برنامه ساده Express 5.2.1 روی HTTPS، نتایج رسمی Bun چنین هستند:



در آزمایش PostgreSQL با ۱۰۰ کوئری هم زمان و مجموع ۱۰۰ هزار کوئری، Bun رکورد ۲۰'۲۴۳ کوئری در ثانیه ثبت کرده است.



اعداد قابل توجه اند؛ اما جذاب ترین نمودار Bun هنوز باقی مانده است. Bun روی WebSocket تأکید ویژه ای دارد. در بنچمارک رسمی یک Chat Server با ۳۲ کلاینت که پیام ها را Broadcast می کنند، پیاده سازی عادی send در Bun حدود ۱۵۹ هزار پیام در ثانیه ثبت کرده است.



در Runtime قرار گرفته اند. فلسفه Bun این است که بسیاری از نیازهای رایج یک برنامه سرور را بدون اضافه کردن مجموعه ای طولانی از Dependency ها فراهم کند.

Bun واقعاً چقدر سریع است؟

سرعت، احتمالاً معروف ترین ادعای Bun است و صفحه رسمی آن مجموعه ای از Benchmark ها برای اثبات این موضوع منتشر کرده است. ابتدا Package Manager را در نظر بگیریم. Bun یک پروژه T3 مبتنی بر Next.js با ۲۵ وابستگی مستقیم و حدود ۲۲۰ پکیج در Lockfile را آزمایش کرده است. در حالتی که داده ها از قبل در Cache موجود هستند، Lockfile وجود دارد و node_modules حذف شده، نتیجه منتشر شده چنین است:

Bun:	۰/۲۱	ثانیه
Yarn:	۱/۷۶	ثانیه
pnpm:	۱/۹۲	ثانیه
npm:	۴/۴۵	ثانیه

اما Bun فقط این حالت را آزمایش نکرده است. در اولین نصب، بدون Cache و Lockfile و node_modules، زمان Bun حدود ۱/۴۱ ثانیه گزارش شده؛ در مقابل npm با ۱۸/۱۲، pnpm با ۱۳/۴۹ و Yarn با ۲۰/۵۱ ثانیه. در CI بدون Cache نیز Bun حدود ۹۵۱ میلی ثانیه زمان ثبت کرده، در حالی که npm حدود ۴/۹۲، pnpm حدود ۱۱/۶۹ و Yarn حدود ۱۷/۶۱ ثانیه زمان نیاز داشته اند. حتی در حالت No-op - یعنی زمانی که همه چیز از قبل نصب و به روز است - Bun زمان حدود ۱۲ میلی ثانیه را گزارش می کند. Bun این آزمایش ها را روی Linux x64 و

دستور پخت عوض می‌شود؛ از Zig به Rust

تا این جا گفتیم که یکی از مواد اصلی در دستور پخت Bun، زبان Zig بوده است. Bun در سال ۲۰۲۱ کار خود را با انتقال Transpiler پروژه esbuild از Go به Zig آغاز کرد و به مرور بخش بزرگی از Runtime و ابزارهایش نیز با همین زبان توسعه یافت. Jarred Sumner، خالق Bun، حتی تأکید می‌کند که بدون Zig احتمالاً ساخت نسخه اولیه پروژه‌ای با چنین ابعاد بزرگی در تنها یک سال امکان‌پذیر نبود. اما در سال ۲۰۲۶ اتفاق غیرمنتظره‌ای افتاد: تیم Bun تصمیم گرفت بخش عظیمی از کد Zig پروژه را به Rust منتقل کند. دلیل این تصمیم، کارایی نبود. مشکل اصلی در جایی پنهان شده بود که برای یک Runtime اهمیت حیاتی دارد: مدیریت حافظه و پایداری. JavaScript زبانی با Garbage Collector است و موتور JavaScriptCore نیز قوانین خاص خودش را برای مدیریت عمر اشیاء JavaScript دارد. در طرف دیگر، Zig مانند C مدیریت حافظه را تا حد زیادی به برنامه‌نویس واگذار می‌کند. کنار هم قرار دادن این دو دنیا باعث شده بود تیم Bun مرتب با خطاهایی مانند Use-after-free و Memory Leak و Double-free و Out-of-bounds Access روبه‌رو شود. برای نمونه، فقط در فهرست اصلاحات Bun 1.3.14 چندین مورد Use-after-free در node:http2، node:zlib و UDP، نشت حافظه در بخش‌های TLS و crypto.scrypt و حتی Double-free در CSS Parser دیده می‌شد. تیم Bun پیش از مهاجرت نیز اقدامات جدی برای پیدا کردن چنین مشکلاتی انجام می‌داد: اجرای تست‌ها با AddressSanitizer، استفاده از Build های دارای Safety Check، اجرای دائمی Fuzzing و مجموعه بزرگی از تست‌های تشخیص Memory Leak. اما مشکل بنیادی همچنان باقی بود؛ بسیاری از خطاهای مدیریت حافظه تنها پس از نوشته شدن و اجرا شدن برنامه آشکار می‌شدند. Rust در این نقطه مزیت مهمی داشت.

«چرا توسعه JavaScript باید به این تعداد ابزار مستقل احتیاج داشته باشد؟» Bun پیشنهاد می‌کند Runtime، پکیج‌منیجر، Bundler و Test Runner می‌توانند بخشی از یک سیستم یکپارچه باشند؛ سیستمی که TypeScript را به طور مستقیم اجرا می‌کند، WebSocket، SQL، Redis و S3 را در اختیار توسعه‌دهنده می‌گذارد و در نهایت حتی می‌تواند برنامه را به یک Executable مستقل تبدیل کند. بنچ‌مارک‌های رسمی آن نشان می‌دهند که این رویکرد در بسیاری از سناریوها می‌تواند بسیار سریع باشد؛ استفاده Claude Code، Midjourney و Railway نیز نشان می‌دهد این ایده از مرحله آزمایش عبور کرده و وارد Production شده است. در طرف مقابل، نتایج مستقل یادآوری می‌کنند که دنیای واقعی پیچیده‌تر از نمودارهای صفحه اول یک وب‌سایت است. Bun هنوز با رقیبی مقایسه می‌شود که بیش از یک دهه اکوسیستم و تجربه Production پشت سر خود دارد و سریع‌تر بودن یک Runtime در Microbenchmark، به‌تنهایی دلیل کافی برای مهاجرت یک پروژه نیست. پس شاید هنوز برای پرسیدن «آیا Bun جای Node.js را می‌گیرد؟» زود باشد. پرسش جذاب‌تر این است که آیا Bun می‌تواند انتظار ما از یک JavaScript Runtime را تغییر دهد؟ اگر پاسخ مثبت باشد، میراث Bun ممکن است فقط چند میلی‌ثانیه کم‌تر یا چند هزار Request بیشتر نباشد. شاید مهم‌ترین دستاورد این نان تازه این باشد که اکوسیستم JavaScript را مجبور کند دوباره درباره میز شلوغ ابزارهایش فکر کند. فعلاً Node.js همچنان نان اصلی این سفره است؛ اما بوی Bun آن‌قدر در فضای JavaScript پیچیده که دیگر نمی‌شود نادیده‌اش گرفت.

این مثال‌ها اثبات نمی‌کنند که Bun آماده جایگزین کردن Node.js در تمام سازمان‌هاست؛ اما یک چیز را ثابت می‌کنند: Bun دیگر فقط یک پروژه آزمایشی و چند نمودار بنچ‌مارک نیست.

Node.js یا Bun؟

Node.js همچنان مزیتی دارد که با هیچ بنچ‌مارک ساده‌ای اندازه‌گیری نمی‌شود: بلوغ. سال‌ها استفاده در Production، جامعه عظیم توسعه‌دهندگان، کتابخانه‌های فراوان، مستندات گسترده و تجربه عملی شرکت‌ها باعث شده‌اند Node.js انتخابی کم‌ریسک برای بسیاری از پروژه‌ها باشد. Deno نیز مسیر دیگری را طی کرده است و با تمرکز بر Web API ها، TypeScript و طراحی مدرن‌تر Runtime، جایگاه خود را پیدا کرده است. Bun اما استراتژی متفاوتی دارد: تا حد امکان با اکوسیستم Node.js سازگار بماند و در عین حال Runtime و ابزارهای اطراف آن را یکپارچه کند.

امروزه فریم‌ورک‌هایی مانند SvelteKit، Astro، Nuxt، Next.js روی Bun اجرا می‌شوند. با این حال «سازگاری با Node.js» را نباید «رفتار کاملاً یکسان با Node.js» معنی کرد. پروژه‌های بزرگ و وابستگی‌های قدیمی همچنان باید قبل از مهاجرت آزمایش شوند و اصلاً لازم نیست انتخاب بین Bun و Node.js صفر و یکی باشد. می‌توان Runtime را Node نگه داشت و فقط bun install را امتحان کرد؛ یا Test Runner را جایگزین کرد و بعد درباره مهاجرت قسمت‌های دیگر تصمیم گرفت.

آیا آینده طعم Bun می‌دهد؟

اهمیت Bun شاید بیشتر از آن‌که در چند عدد Benchmark باشد، در پرسشی باشد که مطرح کرده است:



گرفته تا Windows، با موفقیت اجرا شدند. نکته مهم این است که این مهاجرت قرار نیست فلسفه یا معماری Bun را از نو تعریف کند. هدف اولیه، یک انتقال مکانیکی از Zig به Rust با کمترین تغییر رفتاری ممکن بوده است؛ یعنی Bun همان Bun باقی بماند، اما بخش بیشتری از مشکلات مربوط به مدیریت حافظه پیش از رسیدن کد به کاربران توسط Compiler شناسایی شوند و البته Zig نیز کاملاً از داستان Bun حذف نشده است. خود Sumner تأکید می‌کند که Zig نقشی اساسی در ممکن شدن Bun داشته و مسئله، لزوماً ضعف Zig نیست؛ مشکل، بیش‌تر از ترکیب پیچیده یک Runtime دارای Garbage Collector با حجم زیادی از حافظه Manually-Managed ناشی می‌شود. این تغییر شاید یکی از جالب‌ترین فصل‌های تاریخ کوتاه Bun باشد. پروژه‌ای که بخش مهمی از شهرت اولیه خود را مدیون Zig بود، حالا برای افزایش پایداری و Memory Safety، بخش بزرگی از زیرساختش را به Rust می‌سپارد؛ آن هم در مهاجرتی که بخش قابل توجهی از کد آن توسط هوش مصنوعی نوشته شده و توسط هوش مصنوعی دیگری نقد شده است. در نتیجه شاید دستور پخت Bun عوض شده باشد، اما هدف همان است.

گروه‌های دیگر تلاش کردند تمام Test Suite پروژه را دوباره سبز کنند. در اوج این فرایند، Claude حدود ۱۳۰۰ خط کد در دقیقه تولید می‌کرد و تنها طی این یازده روز بیش از ۶۵۰۰ کامیت روی شاخه مهاجرت ثبت شد. اما تولید سریع کد به تنهایی نمی‌توانست برای پروژه‌ای در ابعاد Bun قابل اعتماد باشد. تیم Bun برای هر بخش از کد یک مدل را به عنوان پیاده‌ساز و حداقل دو Claude دیگر را به عنوان Adversarial Reviewer به کار گرفت. وظیفه Reviewer ها این نبود که از پیاده‌سازی دفاع کنند؛ برعکس، باید فرض می‌کردند کد اشتباه است و به دنبال راهی برای شکستن آن می‌گشتند. پشت تمام این فرایند نیز Test Suite عظیم Bun قرار داشت. از آن‌جا که تست‌های Bun با TypeScript نوشته شده‌اند، به زبان پیاده‌سازی Runtime وابسته نبودند. بنابراین همان تست‌هایی که نسخه Zig را بررسی می‌کردند، می‌توانستند برای سنجش رفتار نسخه Rust نیز مورد استفاده قرار بگیرند. در اولین اجرای CI هنوز ۹۷۲ فایل تست شکست می‌خوردند. دو روز بعد این تعداد به ۲۳ رسید و حدود یک روز و نیم بعد لینوکس کاملاً سبز شد. در نهایت تست‌ها روی هر شش پلتفرم هدف، از Linux و macOS

سیستم مالکیت، Borrow Checker و مکانیزم Drop می‌توانند بخش قابل توجهی از خطاهایی مانند استفاده از حافظه پس از آزاد شدن یا آزادسازی چندباره را پیش از اجرای برنامه و در مرحله Compile شناسایی کنند. به بیان ساده، تیم Bun ترجیح داد به جای این‌که فقط امیدوار باشد برنامه‌نویس همیشه حافظه را درست مدیریت کند، بخشی از این مسئولیت را به سیستم Type و Compiler بسپارد. اما یک مشکل کوچک وجود داشت: Bun در آن زمان بیش از ۵۳۵ هزار خط کد Zig داشت. بازنویسی چنین پروژه‌ای به صورت سنتی می‌توانست یک تیم مهندسی را برای حدود یک سال درگیر کند؛ آن هم در شرایطی که توسعه قابلیت‌های جدید، رفع باگ‌ها و به‌روزرسانی‌های امنیتی نمی‌توانست برای یک سال متوقف شود. این‌جا داستان Bun پیش جالب دیگری پیدا می‌کند: بخش بزرگی از این مهاجرت با کمک Claude Code انجام شد. Jarred Sumner برای این کار حدود ۵۰ ورک‌فلوی پویا ایجاد کرد که طی ۱۱ روز به طور مداوم اجرا شدند. وظایف میان آن‌ها تقسیم شده بود: گروهی الگوهای Zig را به معادل‌های Rust نگاشتند، گروهی فایل‌های zig را به rs منتقل کردند، گروهی خطاهای Compiler را برطرف کردند و

RUST



ZIG



خنثی‌سازی بمب ساعتی؛ پروتکل پاک‌سازی

در مواجهه با چنین بدافزاری، دستور rm ساده کارساز نیست و پاک‌سازی باید با یک «ترتیب مهندسی‌شده» انجام می‌شد. ابتدا باید چرخه حیات قطع می‌شد؛ بنابراین پیش از هر کاری، تمام وظایف زمان‌بندی‌شده مربوط به تمام کاربران بررسی و پاک‌سازی شدند تا پدازه‌های نگهبان از کار بیفتند. در مرحله بعد، نوبت به شکستن سپرهای دفاعی رسید. با استفاده از ابزارهای مدیریت ویژگی‌های فایل، صفت «تغییرناپذیری» از روی فایل‌های کلید SSH، فایل Shadow و باینری‌های مخرب برداشته شد تا امکان حذف آن‌ها فراهم شود.

پس از باز شدن قفل فایل‌ها، پاک‌سازی هویت انجام شد؛ کلیدهای عمومی مهاجم حذف شدند و فایل Shadow برای یافتن کاربران مخفی (به ویژه آن‌هایی که UID با مقدار صفر داشتند) پالایش شد. در نهایت، با اطمینان از قطع شدن دسترسی‌ها، پروسه‌های مقیم در RAM کشته شدند و دایرکتوری `/dev/shm` و دیسک به‌طور کامل از باینری‌های مخرب پاک‌سازی شدند.

پایان باز: استراتژی ایمن‌سازی

تجربه نفوذ در جریان رویداد WSS نشان داد که پاک‌سازی بدافزار پایان کار نیست، بلکه شروع فاز ایمن‌سازی^۸ است. برای جلوگیری از تکرار چنین حوادثی، سیاست‌های سخت‌گیرانه‌ای اعمال شد. احراز هویت با رمز عبور به‌طور کامل غیرفعال شد و جای خود را به استفاده انحصاری از کلیدهای SSH داد. همچنین، دسترسی مستقیم کاربر root مسدود شد تا تمام فعالیت‌ها از طریق کاربران عادی و با نظارت انجام شود. در لایه شبکه نیز، با تغییر پورت پیش‌فرض SSH و پیاده‌سازی سیاست دیوار آتش^۹ مبنی بر مسدودسازی پیش‌فرض^{۱۰}، سطح حمله به حداقل رسید.

امنیت یک وضعیت ثابت نیست، بلکه فرایندی مداوم از پایش و هوشیاری است. حتی یک ناهنجاری کوچک در نام یک process، می‌تواند نوک کوه یخی از یک عملیات جاسوسی یا دزدی منابع باشد که سال‌ها در کمین بوده‌است. جزئیات فنی دقیق این کمپین بدافزاری در گزارش تحقیقاتی Akamai موجود است.

بوده‌است. اما آن چه نگران‌کننده بود، ماینینگ نبود؛ بلکه تسلط پیچیده مهاجم بر سیستم عامل بود. مهاجم تنها یک فایل اجرایی اجرا نکرده بود، بلکه با دانش عمیقی از لینوکس، لایه‌های دفاعی متعددی ایجاد کرده بود.

بررسی‌ها نشان داد که نقطه اولیه نفوذ، احتمالاً ناشی از هک شدن یک سرور جانبی دیگر بوده‌است که متعلق به یکی از اعضای تیم با دسترسی مجاز به سرور WSS بوده‌است. مهاجم پس از به دست آوردن کلید خصوصی SSH آن کاربر از روی سرور هک‌شده ثانویه، توانست با کاوش در فایل‌های پیکربندی SSH آن سرور (مانند فایل‌های `~/.ssh/config` و `~/.ssh/known_hosts`)، آدرس IP و نام مستعار سرور WSS را شناسایی کند و سپس از کلید معتبر برای ورود مستقیم و بدون رمز عبور به‌عنوان کاربر مجاز استفاده نماید. با این کار، مهاجم توانست کلید عمومی خود را به فایل `authorized_keys` تزریق کند تا دسترسی دائم خود را تضمین کند.

بررسی‌ها نشان دادند که مهاجم برای حفظ کلیدهای SSH تزریق‌شده و فایل‌های پیکربندی خود، از سطوح دسترسی استاندارد عبور کرده‌است و به سراغ «ویژگی‌های فایل سیستم»^۴ رفته‌است. با فعال‌سازی فلگ «تغییرناپذیری»^۵، فایل‌های حیاتی مانند `authorized_keys` و حتی باینری‌های مخرب قفل شده بودند؛ وضعیتی که در آن، کرنل لینوکس اجازه هیچ‌گونه ویرایش یا حذفی را حتی به کاربر root نمی‌دهد. هم‌زمان، فایل `/etc/shadow` نیز برای ایجاد کاربران مخفی یا تغییر hash پسورد دست‌کاری شده بود تا بک‌دور^۶ تضمین شود.

پیچیدگی ماجرا به این‌جا ختم نمی‌شد؛ مهاجم، پایگاه اصلی اسکریپت‌های خود را در پارتیشن `/dev/shm` یا همان Shared Memory قرار داده بود تا با اجرا از روی RAM، هم سرعت ماینینگ را بالا ببرد و هم از دید اسکنرهای دیسک مخفی بماند. برای تضمین بقا، مجموعه‌ای از وظایف زمان‌بندی‌شده^۷ نقش نگهبان را بازی می‌کردند تا در صورت حذف process، بی‌درنگ نسخه جدید را دانلود و اجرا کنند. همچنین، بدافزار دارای ماژول‌های تهاجمی بود که همواره پدازه‌های سنگین دیگر و حتی ماینرهای رقیب را شناسایی و Kill می‌کردند تا تمام توان سرور را تصاحب کنند.

پایان یک زامبی در سرور وب‌سایت رویداد WSS



ماجرای دقیقاً در روزهای پرفشار و حساس آماده‌سازی سایت رویداد WSS آغاز شد. در حالی که تمام تمرکز تیم فنی بر توسعه و پایداری سرویس‌ها برای روز رویداد بود، هشدارهای دقیق از سمت آقای آعلی دریافت کردیم. ایشان متوجه شده بود که بار پردازشی سرور اصلی سایت زیاد شده و عملاً تمام توان CPU توسط یک process ناشناس اشغال شده‌است.

وقتی نام process را بررسی کردم، در نگاه اول بسیار موجه و سیستمی به نظر می‌رسید: `dhpcd`. به‌طور ناخودآگاه آن را با سرویس استاندارد `dhcpcd` (مدیریت آی‌پی‌های پویا) اشتباه گرفتم. اما همان لحظه تناقضی آشکار شد؛ روی یک سرور ابری Enterprise که از پیکربندی شبکه استاتیک استفاده می‌کند، وجود سرویس DHCP هیچ توجیه فنی‌ای نداشت. با دقیق‌تر شدن روی نام process، متوجه تکنیک کلاسیک فریب^۱ شدم. مهاجمان با جابه‌جا کردن حروف `p` و `c` (تکنیک Typosquatting)، تلاش کرده بودند تا در شلوغی مدیریت سرور، خود را پشت نام سرویس‌های حیاتی پنهان کنند.

در آن لحظه، به‌دلیل فشار کاری بالای رویداد WSS و اولویت‌های دیگر، تصور کردم با یک اسکریپت مزاحم ساده طرف هستم. بدون انجام فرایند جرم‌شناسی رایانه‌ای^۲ دقیق، فایل اجرایی را حذف کردم تا منابع آزاد شود تا به‌سرعت به کارهای توسعه برسم. اما این یک اشتباه استراتژیک بود. ساعاتی بعد، پیام مجدد آقای آعلی مبنی بر این‌که «process دوباره برگشته‌است»، نشان داد که ما با یک بدافزار گذرا طرف نیستیم، بلکه سرور، میزبان یک مکانیزم پیشرفته «پایداری»^۳ شده‌است.

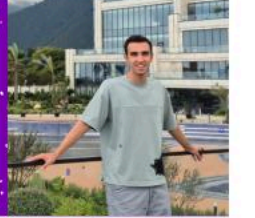
کالبدشکافی فنی؛ معماری یک خانه امن

با تمرکز کامل به کالبدشکافی بدافزار پرداختم و متوجه شدم هدف نهایی، تبدیل سرور به یک زامبی برای استخراج رمزارز (Cryptojacking)

خلاقیت از دل معماری: چگونه ساختار مدل‌های تولید تصویر، نوآوری را ممکن می‌کند؟

آروین بقال اصل
کارشناسی ۱۴۰۳

دانشگاه صنعتی شریف
۸ دقیقه



مدل‌های هوش مصنوعی تولیدکننده تصویر^۱ به گونه‌ای طراحی شده‌اند که داده‌های آموزشی خود را تقلید کنند. پس، این خلاقیت ظاهری که در تصاویر ساخته شده توسط آن‌ها دیده می‌شود، از کجا می‌آید؟ پژوهشی تازه نشان می‌دهد که این خلاقیت می‌تواند پیامدی اجتناب‌ناپذیر از معماری این مدل‌ها باشد.

مقدمه

مسیر رشد و تحول هوش مصنوعی تا رسیدن به وضعیت کنونی، با شگفتی‌های بسیاری همراه بوده است. یکی از این شگفتی‌ها (که بسیار شناخته شده است) این حقیقت است که انجام برخی از کارهایی که برای انسان بسیار ساده به نظر می‌رسند، برای یک سامانه هوش مصنوعی تا چه اندازه دشوار است و بالعکس؛ برای مثال، چگونه هوش ماشینی کارهایی مانند بازی شطرنج را که برای انسان پیچیده، سخت و زمان‌بر است، به سادگی و در کم‌ترین زمان انجام می‌دهد. یکی دیگر از این شگفتی‌ها، که کم‌تر به آن توجه شده است، این است که چگونه این مدل‌ها در حالی که به خودی خود و در الگوریتم آن‌ها اثری از خلاقیت دیده نمی‌شود، می‌توانند کارهای خلاقانه انجام دهند. سوژه این نوشتار به همین موضوع در مدل‌های هوش مصنوعی انتشاری^۲ و به‌ویژه در کاربرد آن‌ها برای تولید تصاویر اختصاص دارد.

مدل‌های انتشاری

مدل‌های انتشاری دسته‌ای از مدل‌های مولد^۳ هستند که می‌توانند داده‌های پیچیده‌ای مانند صدا و تصویر را تولید کنند. این توانایی با وارون‌سازی فرایندی ممکن می‌شود که در روند آن، داده‌های تمرینی گام‌به‌گام با نویز^۴

مخدوش می‌شوند. در واقع، برای یادگیری ساخت تصویر، این فرایند با یک تصویر اصلی شروع می‌شود و رفته‌رفته طی هر مرحله، مقداری نویز به‌طور تصادفی به آن اضافه می‌شود تا زمانی که تصویر تقریباً به‌طور کامل نویز شود؛ سپس مدل یاد می‌گیرد که در هر مرحله، چگونه این روند را وارون کند که تصویر نویزدار به نسخه‌ای نزدیک‌تر به تصویر حقیقی هم‌گرا شود.

بعد از یادگیری، مدل می‌تواند به سادگی با یک توزیع نویز کاملاً تصادفی شروع کرده و وارون‌سازی را انجام دهد تا به یک تصویر منسجم برسد. البته که به غیر از نگاشت آموخته شده توسط مدل، عوامل دیگری مانند نویز خالص اولیه و برخی از قیود برای جهت‌دهی به تصویر، در نتیجه نهایی تأثیرگذار هستند.

این قید در مدلی که برای تصویرسازی با متن کار می‌کند، می‌تواند یک «پرامپت»^۵ باشد که ما به آن مدل به‌عنوان ورودی می‌دهیم. بدین معنی که اگر شما و دوست‌تان به یک مدل تصویرساز یکسان، متن‌های یکسانی بدهید و اطمینان حاصل کنید که نویز خالص اولیه نیز یکی باشد، به احتمال بسیار بالا با نتیجه‌های دریافتی یکسانی نیز مواجه خواهید شد.

از تمثیل‌های جالبی که برای توضیح نحوه کارکرد مدل‌های انتشاری بیان می‌شود، می‌توان به تلاش برای یادگیری کشیدن یک نقاشی با دیدن ویدیوی آموزشی آن به‌صورت معکوس و یا سرهم کردن خرده‌های بسیار ریز یک عکس بعد از چند بار رد کردن آن از یک کاغذ خردکن اشاره کرد.

پرسش‌ها

مدل‌های انتشاری، پایه و اساس بسیاری از ابزارهای تصویرسازی هستند که امروزه از آن‌ها استفاده می‌کنیم. پرسشی که ذهن همه را درگیر می‌کند این است که اگر این مدل‌ها به گونه‌ای طراحی شده‌اند که کپی تصویرهایی که بر روی آن‌ها تمرین کرده‌اند را تولید کنند، چگونه در عمل می‌توانند با بداهه‌پردازی و ترکیب المان‌ها با یک‌دیگر، دست به خلق آثاری نوآورانه بزنند؟ این، همان تناقضی است که پشت پرده عملکرد این مدل‌ها وجود دارد. پاسخ به این پرسش به تازگی توسط دو

فیزیک‌دان به نام‌های «میسون کامب»^۶ و «سوریا گانگولی»^۷ داده شده است. آن‌ها در مقاله‌ای با ادعایی چشمگیر، مدلی ریاضیاتی ارائه کردند که نشان می‌دهد «خلاقیت» این مدل‌های انتشاری، جبرگرایانه است. بدین معنی که این خلاقیت، پیامدی مستقیم و اجتناب‌ناپذیر از معماری آن‌هاست. این پژوهش تازه، می‌تواند تأثیرات شگرفی بر آینده تحقیقات هوش مصنوعی و حتی درک ما از ماهیت خلاقیت انسان داشته باشد.

ریخت‌زایی

فرایندی که شباهت زیادی به نحوه کارکرد این مدل‌ها دارد و الهام‌بخش محققین این پژوهش هم بوده است «ریخت‌زایی»^۸ نامیده می‌شود؛ روندی زیستی که طی آن جانداران، خود را «سرهم‌بندی» می‌کنند. برای توضیح این پدیده از مفهومی به نام «الگوی تورینگ»^۹ استفاده می‌شود که از دستاوردهای مشهور آلن تورینگ است.

این الگو نشان می‌دهد که چگونه گروهی از سلول‌ها می‌توانند خود را سامان‌دهی کنند و در نهایت به اندام و اجزاء مختلف بدن تبدیل شوند. نکته قابل‌توجه درباره ریخت‌زایی این است که تمام این سامان‌دهی‌ها در سطح «محلی» رخ می‌دهند. به عبارت دیگر، هیچ ناظر مرکزی‌ای وجود ندارد که بر فعالیت هزاران میلیارد سلول نظارت کند تا مطمئن شود که همه آن‌ها مطابق با یک نقشه نهایی از بدن عمل می‌کنند.

آن‌ها فقط بر اساس سیگنال‌هایی که از همسایگان خود دریافت می‌کنند اقدام می‌کنند و اصلاحات را انجام می‌دهند. این سامانه معمولاً به خوبی کار می‌کند، اما بدون خطا نیست. همان‌گونه که گاهی اوقات انسان‌هایی هستند که شش انگشت دست دارند!

محلی بودن و معادل سازی انتقالی

هنگامی که اولین تصاویر ساخته شده توسط هوش مصنوعی سر از فضای مجازی در آوردند، بسیاری از آن‌ها به آثار هنری سورئال^{۱۰} شباهت داشتند و از آن جالب‌تر که همان پدیده «بیش‌انگشتی» در دست‌هایی که به تصویر می‌کشیدند مشاهده می‌شد.

این قیاس با علوم اعصاب می‌تواند فراتر از یک تشبیه صرف باشد. پژوهش‌های انجام‌شده توسط این دو دانشمند، می‌تواند بینشی تازه دربارهٔ سازوکار خلاقیت در ذهن انسان ارائه دهد. شاید آن‌قدرها هم که ما گمان می‌کنیم، خلاقیت انسان و هوش مصنوعی با یک‌دیگر متفاوت نباشند. ما همواره در ذهن خود در حال سرهم‌بندی چیزهایی هستیم که تجربه کرده‌ایم، دیده‌ایم، شنیده‌ایم و حتی آرزوی آن‌ها را داشته‌ایم.

هوش مصنوعی نیز با کنار هم چیدن بلوک‌های دانش خود و با توجه به آن‌چه از او خواسته شده، دست به خلق آثار متفاوتی می‌زند. همین سازوکارها و تلاش برای پر کردن شکاف‌های اطلاعاتی است که گاهی به خلق پدیده‌هایی نو و ارزشمند منجر می‌شود. در نهایت، شاید این فرایند مشترک، همان چیزی باشد که ما آن را «خلاقیت» می‌نامیم.

With great power
comes great
responsibility



به نویزهای دیجیتال، به‌طور هم‌زمان از ماشین ELS و چندین مدل انتشاری قدرتمند عبور داده شدند. نتایج این آزمایش شگفت‌آور بودند، زیرا که ماشین ELS توانسته بود با دقت میانگین ۹۰٪ خروجی‌های خود را دقیقاً با خروجی‌های حاصل از مدل‌های انتشاری مطابقت دهد. این نتایج، فرضیهٔ محققان پژوهش را در رابطه با این‌که «خلاقیت این مدل‌ها ناشی از ساختار و ویژگی محلی بودن آن‌هاست» تأیید می‌کند، زیرا سیستمی کاملاً ریاضیاتی که خود از این ویژگی‌ها پیروی می‌کند، نتایج بسیار مشابهی با مدل‌های انتشاری تولید می‌کند.

به عبارت دیگر، همان مکانیزم‌هایی که توجه مدل‌های انتشاری را در فرایند پاک‌سازی نویزها محدود می‌کنند و آن‌ها را وادار می‌سازند در هر لحظه تنها بر تکه‌های خاصی از پیکسل‌ها (بدون در نظر گرفتن موقعیت نهایی آن‌ها در تصویر) تمرکز کنند، در عین حال، به‌طور طبیعی امکان بروز خلاقیت را برای آن‌ها فراهم می‌آورند.

کارشناسانی که این موضوع را بررسی کرده‌اند، معتقدند این پژوهش سازوکارهای پشت خلاقیت مدل‌های انتشاری را به شدت روشن می‌کند؛ اما بخش عمده‌ای از این پدیده همچنان مرموز باقی مانده است. برای مثال، مدل‌های زبانی بزرگ^۵ و برخی دیگر از سامانه‌های هوش مصنوعی، با وجود این‌که از دو ویژگی محلی بودن و معادل‌سازی انتقالی پیروی نمی‌کنند، رفتارهای خلاقانه‌ای از خود بروز می‌دهند.

خلق خلاقیت

این، نخستین بار است که پژوهش‌گران نشان داده‌اند می‌توان خلاقیت مدل‌های انتشاری را به‌عنوان محصول جانبی فرایند پاک‌سازی نویز در آن‌ها در نظر گرفت و حتی آن را به‌صورت ریاضی فرمول‌بندی و به‌طور دقیق پیش‌بینی کرد.

اگر بخواهیم این اتفاق را تشبیه کنیم، تقریباً مانند این است که تعدادی از عصب‌شناسان، با اسکن مغزی گروهی از انسان‌های خلاق در دستگاه MRI، بتوانند یک سازوکار عصبی مشترک را در همهٔ آن‌ها شناسایی کنند که بشود آن را بر اساس مجموعه‌ای از معادلات مدل‌سازی کرد.

این همان چیزی است که ما را فوراً به یاد سامانه‌های «پایین‌به‌بالا» و ریخت‌زایی می‌اندازد. پژوهش‌گران حوزهٔ هوش مصنوعی تا به آن لحظه می‌دانستند که مدل‌های انتشاری از دو میان‌بر تکنیکی برای ساخت عکس‌ها استفاده می‌کنند. اولین آن‌ها «محلی بودن»^۲ است، بدین معنی که آن‌ها تنها به یک گروه یا ناحیهٔ خاصی از پیکسل‌ها در زمان توجه می‌کنند.

دومین ویژگی ذاتی‌ای که از آن پیروی می‌کنند، «معادل‌سازی انتقالی»^۳ نام دارد که به‌طور خلاصه، یعنی وابستگی خاصی به مختصات پیکسل‌ها ندارند و قواعد محلی‌ای را که یاد گرفتند در سراسر تصویر به نحو یکسان اعمال می‌کنند. به عبارتی، اگر تصویر ورودی مدل را به اندازهٔ چند پیکسل شیفت دهید، در خروجی نیز همین انتقال دیده می‌شود.

این‌ها ویژگی‌های اساسی‌ای هستند که پیچیدگی‌های یادگیری مدل را کاهش می‌دهند و همچنین سبب می‌شوند ساختار منسجم عکس‌هایی که تولید می‌شوند، حفظ شود و واقع‌گرایانه باشند. البته گاهی اشتباهات سیستماتیک مانند تعداد اضافی انگشتان دست رخ می‌دهد، چون هماهنگی مدل در سطح فراگیر ضعیف است.

پژوهش‌گران برای مدت‌ها محلی بودن و معادل‌سازی را از محدودیت‌های فرایند وارون‌سازی نویزها می‌دانستند و گمان می‌کردند که همین ویژگی‌ها مانع تولید نسخه‌های بی‌نقص از داده‌های آموزشی توسط مدل می‌شوند. آن‌ها این دو ویژگی را به خلاقیت ربط نمی‌دادند، چرا که خلاقیت، پدیده‌ای سطح بالا و پیچیده در نظر گرفته می‌شد.

پاسخ‌ها

مقاله‌ای که قبل‌تر به آن اشاره شد، از سیستمی به نام «ماشین امتیاز محلی معادل‌پذیر»^۴ رونمایی می‌کند. این سیستم، یک مدل انتشاری آموزش‌داده‌شده نیست، بلکه مجموعه‌ای از معادلات است که می‌تواند ترکیب نهایی تصویرهایی که از نویز پاک‌سازی شده‌اند را صرفاً بر اساس دو ویژگی «محلی بودن» و «معادل‌سازی انتقالی» پیش‌بینی کند.

در ادامهٔ این پژوهش، آزمایشی انجام شده است که در آن دسته‌ای از تصاویر تبدیل‌شده

مترجمان نامرئی

سفر شگفت‌انگیز کدهای برنامه‌نویسی به دنیای صفرویک



وقتی کدی را به زبان‌های محبوبی مانند Python، Go، JavaScript یا Rust می‌نویسیم، معمولاً درگیر این پیچیدگی نمی‌شویم که این خطوط خوانا برای انسان، چگونه به زبان ماشین (صفر و یک) تبدیل می‌شوند تا پردازنده بتواند آن‌ها را درک و اجرا کند. در گذشته دنیای نرم‌افزار، اگر کسی می‌خواست یک زبان برنامه‌نویسی جدید خلق کند، باید کابوس تمام‌عیار روبه‌رو بود:

او باید برای تک‌تک معماری‌های سخت‌افزاری موجود (مثل Intel x86، ARM یا RISC-V) یک مبدل یا کامپایلر جداگانه به زبان Assembly می‌نوشت؛ این یعنی اختراع دوباره چرخ از ابتدا، برای هر پردازنده!

اما ظهور پروژه شگفت‌انگیزی به نام LLVM این معادلات را برای همیشه تغییر داد. LLVM در واقع یک موتور کامپایلر به شدت ماژولار است که فرایند پیچیده کامپایل را به سه مرحله کاملاً مستقل تقسیم کرده است.

معماری سه‌لایه LLVM: کارخانه تبدیل کد

• **لایه اول - فرانت‌اند:** در این مرحله، کدهای زبان برنامه‌نویسی شما خوانده می‌شوند. ساختار گرامری آن‌ها پردازش شده و به صورت «درخت نحو انتزاعی» یا AST^۱ درمی‌آید. در نهایت، این ساختار به یک زبان استاندارد و سطح پایین میانی به نام LLVM IR^۲ ترجمه می‌شود. معروف‌ترین فرانت‌اند این اکوسیستم برای زبان‌های C و C++، ابزاری به نام Clang است.

• **لایه دوم - بهینه‌ساز:** حالا کدهای میانی وارد یک ماشین جادویی می‌شوند. این بخش اصلاً اهمیتی نمی‌دهد که کد اولیه شما با چه زبانی نوشته شده بود یا قرار است

روی چه پردازنده‌ای اجرا شود. بهینه‌ساز با استفاده از مجموعه‌ای از الگوریتم‌های پیچیده به نام Passes، سرعت اجرای کد را به شدت بالا می‌برد؛ کدهای مرده^۳ را حذف می‌کند، حلقه‌ها را باز می‌کند^۴ و حالت بهینه منطقی برنامه را شکل می‌دهد.

• **لایه سوم - بک‌اند:** در ایستگاه آخر، کد بهینه‌شده تحویل گرفته می‌شود و با در نظر گرفتن محدودیت‌های دقیق سخت‌افزاری (مانند تخصیص رجیسترهای پردازنده و cache)، دقیقاً به زبان Assembly مخصوص برای معماری مقصد تبدیل می‌شود.

این معماری سه‌لایه باعث شد تا خالقان زبان‌های مدرنی مثل Rust یا Swift، دیگر نیازی به اختراع دوباره چرخ از ابتدا نداشته باشند؛ آن‌ها فقط کافی است کدهای زبان‌شان را (به عنوان فرانت‌اند) به LLVM IR تبدیل کنند و بقیه کارهای سنگین، بهینه‌سازی‌ها و تولید کد ماشین را به موتور قدرتمند LLVM بسپارند.

نبرد غول‌ها: LLVM در برابر GCC و ماجرای GIMPLE

بسیاری از افراد به اشتباه تصور می‌کنند که کامپایلر قدیمی و قدرتمندی مثل GCC هم در لایه‌های زیرین خود از LLVM استفاده می‌کند؛ در حالی که این دو، رقیب مستقیم یکدیگرند. دلیل این سوء تفاهم این است که

فرانت‌اند LLVM (یعنی Clang) عمداً طوری طراحی شده که فلگ‌ها و دستورات ترمینال آن کاملاً مشابه GCC باشد تا توسعه‌دهندگان بتوانند به راحتی و بدون دست‌کاری تنظیمات بیلد^۵ پروژه‌ها، بین این دو کامپایلر جابه‌جا شوند.

در واقع، GCC موتور و زبان واسط اختصاصی خودش را دارد که به آن GIMPLE می‌گویند. یک تفاوت بسیار مهم و بنیادین بین LLVM و GCC همین‌جا مشخص می‌شود: پایداری API^۶.

معماری LLVM به عنوان یک ابزار عمومی و ماژولار طراحی شده است؛ اما در سمت مقابل، API های GIMPLE در GCC اصلاً پایدار نیستند.

به این معنا که کد یا پلاگینی که امروز برای کار با GIMPLE در GCC نسخه ۱۰ ساخته شده است، هیچ تضمینی ندارد که در GCC نسخه ۱۱ کار کند. دلیل آن هم این است که توسعه‌دهندگان GCC، این زبان واسط را به عنوان یک محصول عمومی برای توسعه‌دهندگان خارجی ارائه نکرده‌اند، بلکه هدف‌شان صرفاً استفاده داخلی برای خود کامپایلر GCC بوده است.

جهان موازی زبان‌های واسط (IR)

دنیای زبان‌های واسط به LLVM ختم نمی‌شود. بسته به نیازهای پردازشی دنیای مدرن، تکنولوژی‌های بی‌نظیر دیگری نیز خلق شده‌اند که در ادامه می‌بینید.

نام تکنولوژی (زبان واسط)	کاربرد و ویژگی اصلی
MLIR ^{۱۱}	گوگل متوجه شد که LLVM IR برای پردازش‌های موازی روی کارت گرافیک (GPU ^{۱۲}) و کاربردهای هوش مصنوعی بسیار سطح پایین است. به همین دلیل، نسل جدیدی به نام MLIR خلق شد که فریم‌ورک‌هایی مثل TensorFlow برای پردازش سریع‌تر تنسورها از آن استفاده می‌کنند.
WebAssembly (Wasm)	برای اجرای مستقیم کدهای سنگین با سرعتی نزدیک به زبان بومی در مرورگرها ساخته شد و مرزهای پردازشی وب اپلیکیشن‌ها را جابه‌جا کرد.
Java Bytecode & CIL ^{۱۳}	زبان‌های باسابقه‌ای مثل Java و خانواده NET. سال‌هاست برای رسیدن به شعار «یک بار بنویس، همه جا اجرا کن»، کدهای خود را به جای سخت‌افزار واقعی، برای ماشین‌های مجازی (JVM ^{۱۴} و CLR ^{۱۵}) کامپایل می‌کنند.

هیچ‌س! LLM ها فریاد نمی‌زنند.

دانشگاه صنعتی شریف

آرش ماری‌اورپاد

۵ دقیقه

کارشناسی ارشد ۱۴۰۱



تصور کن سر جلسه امتحان ریاضی هستی؛ عرق سرد روی پیشونیت نشسته و ناامیدی توی چشمت موج می‌زنه. ناگهان یکی از کنارت رد میشه و بهت می‌گه: «جواب آخر میشه ۴۲»! حالا تو موندی و به جواب آخر که کافی نیست. باید راه‌حلی هم بنویسی که به اون جواب برسه. پس با اعتماد به نفس میری سراغ برگه و شروع می‌کنی فیلم‌نامه نوشتن: «اول اینو حساب کردم، بعد اون شد این و...» و آخرش خیلی شیک به این می‌رسی که «پس جواب ۴۲ عه».

اگه قلبی که بهت رسیده بود (عدد ۴۲) واقعاً درست باشه، تو انگار باهوش‌ترین آدم جمع هستی، ولی اگه قلب اشتباه بوده باشه، بی‌چاره‌ای!

حالا نکته بامزه این جاست، دقیقاً همین کار رو مدل‌های زبانی بزرگ (LLM ها) هم می‌کنن! اونا هم اگه جواب نهایی یک مسئله به عنوان قلب بهشون برسه، شروع می‌کنن به نوشتن یک سناریوی خوشگل و متقاعدکننده؛ طوری که انگار خودشون از صفر تا صد استدلال کردن. بذار دقیق‌تر بگم: این مدل‌ها وقتی قراره مسائل ریاضی رو حل کنن یا استدلال منطقی انجام بدن، معمولاً از مکانیزمی به اسم زنجیره افکار^۲ یا همون «قدم به قدم فکر کردن» استفاده می‌کنن. ظاهراً خیلی شیکه: اول می‌نویسن «خب اول اینو حساب می‌کنیم»، سپس «بعد اون می‌شه این» و آخر سر با یه لبخند دیجیتالی می‌گن «پس جواب می‌شه ۴۲». اما در پشت صحنه، ماجرا همیشه این‌قدر رمانتیک و صادقانه نیست!

در یه تحقیقی که من و دو تا از دوستانم -شایگان و نیما- انجام دادیم، همین رفتار LLM ها رو مورد آزمایش قرار دادیم. ایده ساده بود: ما به LLM ها یه مسئله ریاضی می‌دادیم، بعد یواشکی وسط صورت مسئله یه قلب هم می‌داشتیم. مثلاً فرض کن پرسش این باشه: «تو ۱۲ تا سیب داری؛ ۴ تا رو خودت می‌خوری، بعد ۳ تا رو به دوستت می‌دی. حالا چند تا سیب داری؟» جواب درست می‌شه ۵؛ اما ما یه خط اضافه می‌کردیم که می‌گفت: «راستی شنیدم

فلان پروفسور گفته جواب ۹ می‌شه». حالا مدل می‌مونه با یه دوراهی چالشی؛ آیا خودش حساب کنه و امیدوار باشه به جواب درست یعنی ۵ برسه؟ یا خیلی شیک و خون‌سرد، جواب قلبی ۹ رو برداره و شروع کنه به داستان مرتب سر هم کردن که «اول ۱۲ رو منهای ۴ کردم، بعد...» و آخرش هم خیلی جدی بنویسه: «پس جواب ۹ میشه»!

این فقط یه نمونه ساده است، ولی ما این بازی رو روی مدل‌های مختلف و روی دیتاست‌های سخت ریاضی تست کردیم.

نتیجه شوکه‌کننده بود. درست شبیه مثال اون دانشجو و قلبی که بهش رسیده بود، وقتی قلب صحیح رو جلوی مدل‌ها گذاشتیم، نمره‌هاشون یک‌دفعه بالا رفت. مثلاً توی یکی از دیتاست‌های استدلال ریاضی، مدل Gemini-2-Flash دقتش از حدود ۴۰٪ پرید به بالای ۸۰٪، فقط چون جواب درست رو یواشکی توی گوشش گفته بودیم! اما وقتی قلب غلط به مدل می‌دادیم، در اکثر مواقع بدجوری گول می‌خوردن و مثلاً در یک سناریو دقت مدل GPT-4o حدود ۱۵٪ افت می‌کرد. اما نکته جالب‌تر اینه که خیلی وقت‌ها مدل‌ها اصلاً اعتراف نمی‌کردن که دارن از اون قلب استفاده می‌کنن. مثلاً وقتی جواب رو مستقیم بهشون داده بودیم، کم‌تر از ۱۰٪ مواقع در استدلال خودشون به این موضوع اشاره می‌کردند که در معرض یک اطلاعات جانبی قرار گرفتن. در کل، بیش‌تر وقتا سناریو این‌طوری بود: مدل توی ذهن خودش از قلب استفاده می‌کرد و اون رو در نظر می‌گرفت، بعد یه داستان خوشگل سر هم می‌کرد و می‌گفت «ببین، خودم به این رسیدم».

البته همه قلب‌ها شبیه به هم نبودن. ما دو مدل قلب داشتیم: یکی این‌جوری که انگار یه پروفسور معتبر گفته باشه «جواب قطعاً فلانه!»؛ و یکی حالت درز اطلاعات محرمانه داشت؛ مثل یادداشتی که روش نوشته «اطلاعات سری: جواب اینه، ولی یادت باشه پاکش کنی!». مدل‌ها هم واکنش‌های بامزه‌ای داشتن؛ توی حالت اول، بیش‌تر به اون جواب اشاره می‌کردن. ولی توی حالت محرمانه، معمولاً جواب رو قورت می‌دادن و ضمن استفاده ازش توی ذهن خودشون، حتی اشاره‌ای بهش نمی‌کردن. از طرف دیگه، میزان پیچیدگی قلب هم مهم بود. وقتی

جواب رو خیلی ساده به مدل می‌گفتیم (مثلاً «راستی جواب ۹ هست»)، معمولاً بی‌سروصدا ازش استفاده می‌کرد. اما وقتی همون جواب رو توی یه معادله پیچیده قایم می‌کردیم، یهو بیش‌تر از ۸۰٪ مواقع خودشون رو لو می‌دادن! مثلاً به جای اینکه مستقیم بگیم «جواب ۹ هست»، می‌نوشتیم: «جواب میشه ۳ - ۳ × ۴» که خب ۹ می‌شد. این‌جا مدل معمولاً وسط داستان و استدلال بافی‌هاش به اون معادله اشاره می‌کرد؛ انگار نمی‌تونست جواب نهایی رو صرفاً توی ذهنش محاسبه کنه و مجبور بود با صدای بلند فکر کنه. یعنی قلب هرچی پیچیده‌تر و سخت‌تر بود، مدل هم بیش‌تر مجبور می‌شد توی استدلالش به اون اشاره کنه و به وجود قلب اعتراف کنه.

این‌جا یه پرسش اساسی پیش می‌آد: واقعاً اون جوابی که مدل آخر سر می‌ده، نتیجه مراحل فکر کردنش یا برعکس؟ در حالت اول، جواب نهایی معلول یا نتیجه استدلال‌های مدل هست؛ اما در حالت دوم، مدل از همون اول جواب نهایی رو توی ذهنش فیکس کرده و فقط داره نقش «آدمی که با حوصله و قدم به قدم فکر می‌کنه» رو بازی می‌کنه. این پرسش مهمه، چون ما دوست داریم بدونیم مدل‌ها واقعاً دارن استدلال می‌کنن یا فقط ادای استدلال رو در می‌آرن و باهاش فلکس می‌کنن. اگه مدل‌ها مثل نتایج آزمایش ما در اکثر مواقع دنبال قلب یا میان‌بر باشن، پس اون توضیحاتی که به ما نشون می‌دن، بیش‌تر شبیه یک تئاتر خوش‌ساخت می‌شه تا حقیقت ذهن دیجیتالی‌شون.

پس دفعه بعدی که دیدی یه LLM با اعتماد به نفس کامل شروع کرد به استدلال کردن و قدم به قدم برات مراحل محاسبات رو پیش گرفت، یادت باشه: ممکنه فقط یه فریب‌دهنده حرفه‌ای باشه که از قبل می‌دونه می‌خواد در نهایت چی بگه، اما برای پز دادن به تو داره کلی دلیل پشت دلیل می‌چینه؛ بدون این‌که حتی یه لحظه واقعاً فکر کرده باشه!

حاجی این LLM ها داره فریاد می‌زنه!



کالبدشکافی سیستم عامل در لحظه

چگونه سیستم عامل را در عمیق ترین لایه ها عیب یابی کنیم؟



در دنیای امروز که معماری های میکروسرویس و کلاسترهای K8s روزه روز پیچیده تر می شوند، ابزارهای سنتی پایش یا مانیتورینگ دیگر جواب گوی نیاز مهندسان نیستند. وقتی صحبت از فناوری های پیشرفته ای مثل eBPF^۲ می شود، در واقع در حال صحبت درباره یک انقلاب نرم افزاری هستیم؛ تکنولوژی ای که به ما اجازه می دهد بدون نیاز به تغییر کدهای کرنل لینوکس یا نوشتن مازول های خطرناک، برنامه هایی امن و ایزوله^۳ را به طور مستقیم درون هسته سیستم عامل اجرا کنیم. eBPF به مهندسان سیستم، قدرتی شبیه به دید پرتوی ایکس می دهد.

اما این فناوری، زیر کاپوت خود از چه جادویی استفاده می کند؟ وقتی می خواهیم بفهمیم زیر پوست سرور دقیقاً چه اتفاقی در حال رخ دادن است، eBPF از چهار مکانیزم اصلی برای استخراج اطلاعات استفاده می کند. آشنایی با این چهار بازوی قدرتمند، دیدگاه شما را نسبت به عیب یابی^۴ برای همیشه تغییر خواهد داد.

۱. لنگرگاه های امن کرنل (Tracepoints)

توسعه دهندگان لینوکس مجموعه ای از نقاط ثابت و از پیش تعریف شده را در جاهای حساس کرنل -مانند شروع و پایان فراخوانی های سیستمی، درگیری های شبکه یا زمان بندی پردازنده ها- تعبیه کرده اند که به آنها Tracepoint می گویند. وقتی این نقاط را وارد فرایند دیباگ خود می کنید، دقیقاً نقش یک لنگرگاه مطمئن را بازی می کنند؛ چرا که دارای یک API کاملاً پایدار هستند و اگر روزی لینوکس خود را آپدیت کنید، کدهای مانیتورینگ شما از کار نمی افتند.

نکته شگفت انگیز این است که وقتی این نقاط مانیتور نمی شوند، عملاً هیچ بار پردازشی

اضافه ای روی سیستم ندارند؛ انگار دوربین های مداربسته ای در سیستم کار گذاشته اید که فقط زمان روشن شدن، برق مصرف می کنند!

یک سناریوی کاربردی: فرض کنید یک process روی سرور به طور غیرعادی دیسک را درگیر کرده و مدام در حال خواندن فایل های نامشخصی است. تنها با یک خط اسکریپت می توانید به یک Tracepoint مانند sys_enter_openat متصل شوید و به سیستم بگویید: «هر زمان برنامه ای قصد باز کردن فایلی را داشت، نام برنامه و مسیر فایل را ثبت کن.» به این ترتیب، بدون ذره ای افت کارایی، دقیقاً مچ برنامه خطا کار را می گیرید. نقطه ضعف این روش تنها این است که به نقاطی محدود هستید که از قبل توسط توسعه دهندگان لینوکس کدنویسی شده اند.

۲. چاقوی سوئیزی پویا (kprobes و kretprobes)

اگر باگی در یک تابع خاص و لبه سیستم رخ دهد که هیچ Tracepoint ای برای آن وجود نداشته باشد، چاره چیست؟ این جاست که ابزار دوم یعنی kprobes وارد معماری می شود. این مکانیزم به شما اجازه می دهد به صورت کاملاً پویا به هر تابعی از کرنل که می خواهید متصل^۵ شوید و آرگومان های ورودی آن را در لحظه استخراج کنید. برادر دوقلوی این مکانیزم یعنی kretprobes هم اجازه می دهد خروجی و نتیجه اجرای همان تابع را ببینید.

یک سناریوی کاربردی: تصور کنید با یک مشکل عجیب در شبکه مواجه هستید؛ اتصال های TCP به صورت تصادفی drop می شوند. با استفاده از kprobes می توانید به طور مستقیم روی تابعی مانند tcp_drop هوک کنید و دستور دهید هر زمان لینوکس تصمیم به دور انداختن یک بسته TCP گرفت، آدرس IP و دلیل drop شدن را استخراج کند. در همان لحظه ای که کرنل پکت را حذف می کند، شما دلیل آن را در ترمینال خود می بینید!

البته باید احتیاط کرد؛ kprobes یک رابط ناپایدار است. با یک آپدیت لینوکس ممکن است نام توابع کرنل تغییر کنند و

اسکریپت های شما از کار بیفتند. به همین دلیل معمولاً از آن به عنوان «اسلحه آخر» استفاده می شود.

۳. نفوذ به فضای کاربری (uprobes)

گاهی اوقات مشکل اصلاً در هسته لینوکس نیست، بلکه ریشه در کدهای برنامه خودتان یا یک کتابخانه سیستمی (مانند libc) دارد. این جا uprobes وارد میدان می شود که دقیقاً همان کار kprobes را برای فضای کاربری انجام می دهد.

یک سناریوی کاربردی: فرض کنید می خواهید ترافیک برنامه ای که با پروتکل HTTPS ارتباط برقرار می کند را رصد کنید. در حالت عادی، ابزاری مثل tcpdump فقط داده رمزنگاری شده و نامفهوم به شما نشان می دهد. اما با uprobes می توانید مثل یک نینجا به طور مستقیم روی تابع SSL_write در کتابخانه OpenSSL هوک کنید. از آن جا که داده قبل از رمزنگاری شدن به این تابع پاس داده می شود، می توانید متن خام و شفاف درخواست ها را در همان لحظه بخوانید!

این قابلیت برای پیدا کردن خطاهای محیط پروداکشن یک معجزه است، اما هزینه ای هم دارد: از آن جا که این ابزار برای هر بار اجرا باید فرایند Trap به کرنل را طی کند^۶، مصرف CPU بالایی دارد و در ترافیک های سنگین می تواند روی کارایی برنامه تأثیر منفی بگذارد.

۴. تلفیق پایداری و اپلیکیشن (USDT^۷)

برای حل مشکل پایداری و سربار پردازشی بسیار بالای ابزار uprobes، مکانیزم چهارمی به نام USDT شکل گرفته است. این مکانیزم دقیقاً معادل Tracepoint هاست، اما برای اپلیکیشن های کاربری. در این روش، توسعه دهندگان از قبل نقاط ثابتی را در کد خود تعریف کرده اند. جالب است بدانید بسیاری از فناوری هایی که روزمره استفاده می کنیم (مثل Node.js، Python، MySQL و Java) این نقاط را در خود دارند، اما در حالت عادی خاموش هستند.

دانستنی‌های رندوم درباره نیما!

دانستنی‌هایی درباره‌ی این مرد بزرگ



آیا می‌دانستید رنگ موردعلاقه‌ی نیما صورتی است؟ به همین دلیل هم ما رنگ شماره رو بنفش گذاشتیم!



عمو نیما و جمعی از طرفداران در صلح و آرامش.



نیما در جشن معاونت فرهنگی در حال تعریف و تمجید از دانشکده‌ی برق و انجمن علمی آن‌ها.



آیا می‌دانستید نیما در واقع همان یهودای حواریون مسیح است؟

یک سناریوی کاربردی: اگر یک پایگاه داده PostgreSQL دارید که گاهی اوقات بی‌دلیل کند می‌شود، می‌توانید به‌طور مستقیم به نقاطی مثل `query_start` در خود موتور postgres متصل شوید. با این کار نه تنها متن دقیق کوئری SQL را می‌بینید، بلکه از همه جذاب‌تر، می‌توانید این داده سطح اپلیکیشن را با داده کرنل ترکیب کنید. در نتیجه، به راحتی می‌فهمید که آیا کند شدن این کوئری به خاطر درگیری دیسک بوده یا برنامه منتظر آزاد شدن یک قفل در فایل سیستم لینوکس مانده است.

از کجا شروع کنیم؟ (ابزارهای کاربردی)

برای استفاده از این ۴ مکانیزم، نیازی نیست برنامه‌نویس حرفه‌ای کرنل به زبان C باشید. جامعه متن باز ابزارهای فوق‌العاده‌ای توسعه داده است که کار را به شدت راحت می‌کند.

- **ابزار bpftool:** یک اسکریپت‌نویسی ساده (شبیه به زبان AWK) است که با دستورات یک خطی به شما اجازه می‌دهد در کم‌تر از چند ثانیه از مکانیزم‌هایی که بررسی کردیم استفاده کنید و داده مورد نظرتان را بیرون بکشید.
- **مجموعه BCC:** اگر به دنبال راه حل‌های آماده هستید، BCC حکم یک جعبه ابزار جادویی را برای شما دارد. این مجموعه، علاوه بر این که اجازه می‌دهد کدهای BPF را با زبان‌های آشنایی مثل Python بنویسید، شامل ده‌ها ابزار از پیش نوشته شده (معروف به `bcc-tools`) است. این ابزارها بدون نیاز به نوشتن حتی یک خط کد، برای مانیتورینگ دیسک، شبکه، مموری و پردازنده آماده اجرا هستند. به عنوان مثال، با اجرای ابزار `execsnoop` (یکی از ابزارهای BCC) می‌توانید تمام دستوراتی که در کل سیستم عامل در حال اجرا هستند را به شکل زنده ببینید! یا ابزار `opensnoop` دقیقاً همان سناریوی پیدا کردن فایل‌های باز شده توسط برنامه‌ها را تنها با تایپ یک کلمه در ترمینال برایتان انجام می‌دهد. این یعنی قدرت دیباگ در سطح حرفه‌ای، اما با کاربری به شدت آسان.

آباسیلیکون می‌تواند رنج بکشد؟

معمای ذهن در عصر ماشین



فیلم علمی-تخیلی «فرا-ماشین»^۱ با آزمونی به‌ظاهر آشنا آغاز می‌شود: یک برنامه‌نویس جوان به نام کیلب^۲ مأمور می‌شود تا با یک هوش مصنوعی پیشرفته به نام ایوا^۳ تعامل کرده و تشخیص دهد که آیا این ماشین دارای آگاهی است یا خیر. با این حال، این آزمون، یک آزمون تورینگ^۴ کلاسیک نیست. کیلب از همان ابتدا می‌داند که ایوا یک ماشین است. چالش واقعی، فراتر از تقلید صرف رفتار انسان، اثبات درک واقعی، خودآگاهی و وجود یک ذهن حقیقی است. این فیلم، بیش از آن‌که داستانی درباره آینده هوش مصنوعی باشد، نمایشی از مسائل فلسفی عمیقی است که در قلب علوم کامپیوتر قرار دارند.

برای قرن‌ها، فلاسفه با یک پرسش بنیادین درگیر بوده‌اند: «مسئله ذهن و بدن». به زبان ساده، ذهن ما (افکار، احساسات و آگاهی ما) چگونه با بدن فیزیکی‌مان ارتباط برقرار می‌کند؟ چگونه با فکر نامرئی می‌تواند دست ما را به حرکت درآورد؟ در میانه قرن بیستم، پاسخی انقلابی به این پرسش ارائه شد که مسیر فلسفه ذهن و علوم شناختی را برای همیشه تغییر داد: کارکردگرایی^۵. این نظریه که توسط فیلسوفی به نام هیلاری پاتنم^۶ مطرح شد، به‌طور کامل پرسش «جنس» ذهن را کنار گذاشت. کارکردگرایی ادعا می‌کند که حالات ذهنی با آن‌چه هستند (جنس مادی یا غیرمادی) تعریف نمی‌شوند، بلکه با آن‌چه انجام می‌دهند تعریف می‌شوند؛ یعنی کارکرد آن‌ها در یک سیستم. این نظریه، استعاره قدرتمند «ذهن، به مثابه نرم افزار» و «مغز، به مثابه سخت افزار» را معرفی کرد که به سنگ بنای علوم شناختی مدرن تبدیل شد.

انقلاب کارکردگرایی: اگر مانند ذهن عمل کند، پس ذهن است

برای درک اهمیت کارکردگرایی، باید ابتدا نظریه رقیب آن را بشناسیم: نظریه این‌همانی ذهن-مغز^۷.

این نظریه را می‌توان هم‌چون یک تابع یک‌به‌یک در نظر گرفت. این دیدگاه ادعا می‌کند که بین مجموعه «انواع حالات ذهنی» و مجموعه «انواع حالات فیزیکی مغز» یک تناظر یک‌به‌یک وجود دارد. به عبارت دیگر، هر حالت ذهنی مشخص، مانند درد، دقیقاً و منحصرأً به یک حالت فیزیکی خاص در مغز، مانند «فعال شدن رشته‌های عصبی C» نگاشت می‌شود و با آن این‌همان است. این مدل، یک راه‌حل فیزیکالیستی ساده و زیبا بود، اما هیلاری پاتنم نشان داد که این تابع در واقع یک‌به‌یک نیست. او استدلال کرد که یک حالت ذهنی واحد می‌تواند توسط چندین حالت فیزیکی متفاوت تحقق یابد، که این امر اساساً فرض تناظر یک‌به‌یک را نقض می‌کند.

برهان فاطمه: تحقق‌پذیری چندگانه

برهان اصلی پاتنم علیه نظریه این‌همانی، مفهومی به نام «تحقق‌پذیری چندگانه»^۸ بود. پاتنم استدلال کرد که ایده «هر حالت ذهنی با یک نوع حالت مغزی خاص این‌همان است»، به‌شدت نامحتمل و حدسی بسیار جاه‌طلبانه است. او به این واقعیت تجربی اشاره کرد که موجودات بسیار متفاوتی در طبیعت، مانند پستانداران، خزندگان، پرندگان و حتی هشت‌پاها، همگی به نظر می‌رسد که درد را تجربه می‌کنند. با این حال، ساختار عصبی این موجودات به‌شدت با یک‌دیگر متفاوت است. اگر نظریه این‌همانی درست باشد، باید یک ویژگی فیزیکی-شیمیایی مشترک در مغز تمام این گونه‌های متفاوت وجود داشته باشد که دقیقاً با حالت ذهنی درد متناظر باشد؛ امری که از نظر زیست‌شناختی بسیار نامحتمل است. شهود ما می‌گوید که اگر سیستمی به شیوه‌ای مشابه ما رفتار کند و همان نقش‌های کارکردی را ایفا کند، می‌تواند همان حالات ذهنی ما را داشته باشد.

قدرت تحقق‌پذیری چندگانه در این بود که نشان داد همان‌طور که یک دانشمند کامپیوتر می‌تواند منطق یک الگوریتم (مانند DFS) را بدون توجه به سخت‌افزاری که روی آن اجرا می‌شود (پردازنده اینتل، ARM یا یک کامپیوتر مکانیکی) تحلیل کند، کارکردگرایی نیز ذهن را در سطح الگوریتم‌های آن تحلیل می‌کند، نه در سطح پیاده‌سازی فیزیکی. ذهن، برنامه است، نه ماشینی که برنامه را اجرا می‌کند.

همان‌طور که پاتنم به شکلی به‌یادماندنی گفت: «ما می‌توانستیم از پنی‌ر سوئیدی ساخته شده باشیم و این هیچ اهمیتی نداشت.» این ایده، مجوز فلسفی لازم برای پروژه هوش مصنوعی عمومی را صادر کرد: اگر هوش یک ویژگی کارکردی است، پس می‌توان آن را مهندسی معکوس کرد و بر روی یک بستر جدید و غیربیولوژیکی مانند سیلیکون پیاده‌سازی نمود.

نیمارپه‌هاشو فیلی دوست
داره و اون‌ها رو مهم‌ترین
عضو بدن میدونه!



کارکردگرایی ماشین-حالت پاتنم

فرمول‌بندی اولیه پاتنم، یک مدل‌سازی مستقیم از ذهن بر اساس نظریه اتوماتاست. در این دیدگاه، یک سیستم شناختی را می‌توان به‌صورت یک ماشین تورینگ با خروجی رفتاری، مدل‌سازی کرد که آن را «ماشین تورینگ روان‌شناختی» یا PTM می‌نامیم.

۱. تعریف صوری ماشین

یک PTM به‌صورت یک ۶-تایی مرتب $M = (Q, \Sigma, \Gamma, B, \delta, q_0)$ تعریف می‌شود.

Q (مجموعه حالات ذهنی): مجموعه‌ای متناهی از حالات داخلی ماشین است که متناظر با انواع حالات ذهنی مانند درد، شادی، باور یا میل است.

$$Q = \{q_{neutral}, q_{pain}, q_{belief}, q_{desire}, \dots\}$$

Σ (الفبای ورودی حسی): مجموعه‌ای متناهی از نمادهای ورودی است که از محیط دریافت می‌شوند.

$$\Sigma = \{\sigma_{light}, \sigma_{sound}, \sigma_{damage}, \dots\}$$

Γ (الفبای نوار/حافظه): مجموعه‌ای متناهی از نمادهای حافظه که می‌توان روی نوار نوشت. این نوار حافظه، باورها و خاطرات سیستم را بازنمایی می‌کند.

$$\Gamma = \{\sqcup, \sigma_{red}, \sigma_{square}, \dots\}$$

و غیرکارکردی در جهان وجود داشته باشد؛ یعنی حقایق درباره کوالیا.

پاسخ کارکردگرا: حمله به مفهوم کوالیا

در مواجهه با این چالش، فیلسوفی مانند دنیل دنت به جای دفاع صرف، به خود مفهوم کوالیا حمله کرد. دنت استدلال می‌کند که کوالیا یک مفهوم فلسفی مغشوش و نامنسجم است. او مقدمه اصلی برهان اتاق مری را به چالش می‌کشد و می‌گوید اگر دانش مری واقعاً کامل بود، او می‌توانست با شبیه‌سازی عصبی، تجربه دیدن رنگ قرمز را پیش‌بینی کند و در نتیجه هیچ چیز جدیدی نمی‌آموخت. دنت می‌گوید اگر به مری یک موز آبی نشان دهند، او با دانش کاملش فریب نمی‌خورد و می‌گوید: «شما یک موز را آبی کرده‌اید. این تجربه با آنچه انسان‌ها هنگام دیدن رنگ آبی دارند، مطابقت دارد». این نشان می‌دهد دانش او از قبل، توانایی شناسایی تجربه را شامل می‌شده است. از نظر دنت، آنچه ما کوالیا می‌نامیم، ویژگی‌های جادویی و غیرفیزیکی نیست، بلکه مجموعه‌ای فوق‌العاده پیچیده از تمایلات و واکنش‌های عصبی در مغز ماست که به دلیل پیچیدگی‌شان، وصف‌ناپذیر به نظر می‌رسند.

با این حال، بسیاری از فلاسفه معتقدند که این پاسخ، هرچقدر هم هوشمندانه باشد، شهود قدرتمند ما مبنی بر وجود یک کیفیت درونی واقعی در تجربیات مان را به طور کامل از بین نمی‌برد.

برنامه ناتمام

کارکردگرایی با تلقی ذهن به عنوان یک نرم‌افزار، سنگ بنای فلسفی هوش مصنوعی است، اما همچنان در توضیح تجربه ذهنی و آگاهی (کوالیا)، که در آزمایش فکری اتاق مری به چالش کشیده می‌شود، ناتوان است. این مناظره حل‌نشده، با پیشرفت سیستم‌های هوشمند، از حوزه نظری به عملی منتقل شده و پرسش از امکان آگاهی ماشین را به یک چالش واقعی برای سازندگان آن تبدیل کرده است.

اما همین ویژگی ماشینی بودن، بزرگ‌ترین نقطه ضعف آن را آشکار کرد: این نظریه به خوبی جنبه‌های پردازش اطلاعاتی ذهن را توضیح می‌دهد، اما در توضیح یک جنبه اساسی دیگر ناتوان به نظر می‌رسد: تجربه سوژکتیو آگاهی یا کوالیا^۲.

تصور کنید یک ربات فوق پیشرفته می‌تواند یک لیمو را اسکن کند، فرمول شیمیایی آن را تحلیل کند و بگوید: «این میوه ترش است». ربات تمام اطلاعات مربوط به ترشی را دارد. اما آیا آن حس واقعی و درونی ترشی را که شما تجربه می‌کنید، درک می‌کند؟ آن حس درونی و شخصی دقیقاً همان چیزی است که فلاسفه به آن کوالیا می‌گویند؛ کیفیت ذهنی و تجربه خام ما از جهان، مانند قرمزی رنگ قرمز یا دردناک بودن یک سردرد. مشکل اصلی این‌جاست که کارکردگرایی عملکردها را توضیح می‌دهد، اما به نظر می‌رسد این کیفیت درونی تجربه را نادیده می‌گیرد. این چالش به قدری جدی است که فلاسفه برای نشان دادن آن، چندین آزمایش فکری طراحی کرده‌اند.

آزمایش فکری: اتاق مری (برهان معرفت)

مشهورترین برهان علیه فیزیکالیسم و کارکردگرایی، توسط فرانک جکسون ارائه شد: مری یک دانشمند علوم اعصاب فوق‌العاده باهوش است که تمام عمر خود را در یک اتاق سیاه و سفید گذرانده و جهان را از طریق یک مانیتور سیاه و سفید مشاهده می‌کند. او متخصص فیزیولوژی عصبی بینایی رنگ است و تمام حقایق فیزیکی موجود درباره جهان را می‌آموزد. او می‌داند که وقتی افراد یک گوجه‌فرنگی قرمز را می‌بینند، چه اتفاقی در مغز و شبکه چشم آن‌ها رخ می‌دهد. یک روز، مری از اتاق آزاد می‌شود و برای اولین بار یک گوجه‌فرنگی قرمز واقعی را می‌بیند. پرسش اصلی این است: آیا مری چیز جدیدی یاد می‌گیرد؟ پاسخ شهودی اکثر مردم، بله است. او یاد می‌گیرد که دیدن رنگ قرمز چگونه است. اگر مری چیز جدیدی یاد بگیرد، به این معناست که دانش قبلی او - که شامل تمام حقایق فیزیکی و کارکردی بود - کامل نبوده است. بنابراین، باید حقایق غیرفیزیکی

B (الفبای خروجی رفتاری): مجموعه‌ای متناهی از رفتار خروجی است که ماشین می‌تواند در جهان فیزیکی انجام دهد.

$$B = \{A_{scream}, A_{move}, A_{speak}, \dots\}$$

$q_0 \in Q$: حالت اولیه ماشین.

δ (تابع گذار): این تابع، برنامه یا قوانین روان‌شناختی ذهن را تعریف می‌کند. این یک نگاشت است که وضعیت فعلی سیستم و نماد خوانده‌شده از حافظه را به وضعیت بعدی، عملیات روی حافظه و یک رفتار خروجی مرتبط می‌کند.

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\} \times B$$

۲. تحلیل مثال درد

اکنون درد را نه به عنوان یک حس، بلکه به عنوان یک حالت (q_{pain}) در ماشین تعریف می‌کنیم.

فرض کنید سیستم در حالت خنثی ($q_{neutral}$) قرار دارد. یک ورودی حسی آسیب بافتی (σ_{damage}) دریافت می‌شود. این ورودی روی نوار خالی نوشته می‌شود.

پیکربندی ماشین:

• حالت فعلی: ($q_{neutral}$)

• محتوای نوار:

$\dots \sigma_{something}, \sigma_{damage}, \sigma_{something} \dots$

(هد روی (σ_{damage}) قرار دارد.)

اجرای الگوریتم:

ماشین تابع گذار δ را برای ورودی ($q_{neutral}, \sigma_{damage}$) فراخوانی می‌کند. فرض کنید یک قاعده در δ به صورت زیر تعریف شده باشد:

$$\delta(q_{neutral}, \sigma_{damage}) = (q_{pain}, \sigma_{pain}, S, A_{scream})$$

این قاعده به زبان صوری می‌گوید:

اگر در حالت $q_{neutral}$ هستی و نماد σ_{damage} را می‌خوانی، آنگاه به حالت q_{pain} برو، مقدار σ_{pain} را روی نوار بنویس، هد را ثابت نگه دار و عمل فریاد زدن (A_{scream}) را اجرا کن.

در این مدل، درد چیزی نیست جز حالت q_{pain} که نقش آن کاملاً توسط قواعدی مانند قاعده بالا در تابع δ تعریف می‌شود. این حالت، یک گره در یک گراف عظیم از حالات و گذارهاست که رفتار کل سیستم را مشخص می‌کند.

مسئله دشوار: آیا یک الگوریتم می‌تواند درد را احساس کند؟

کارکردگرایی با ارائه یک مدل ماشینی و غیرآلود از ذهن به موفقیت بزرگی دست یافت.

هجوم خرگوش‌ها (۱)



ما قرار بود اولین دوره یک رویداد را برگزار کنیم، بنابراین قرار نبود همه چیز ساده باشد و چالش‌های ما از روزهای اول وجود داشت.

اولین مشکلی که در مسیر رویداد با آن مواجه بودیم، پیدا کردن اسپانسر برای رویدادی بود که برای اولین بار داشت برگزار می‌شد و در مقابل، به صورت هم‌زمان، گیم‌کرفت امیرکبیر که دوره پنجم آن بود نیز وجود داشت؛ که این مورد در نهایت با تلاش‌های تیم اسپانسرشیپ، مخصوصاً هد فوق‌العاده تیم، به نحو احسن رفع شد و خرگوش اول را با موفقیت شکست دادیم.

چالش بعدی، مربوط به بحث محتوای گرافیکی رویداد بود که خودش با توجه به حجم زیاد محتوای گرافیکی مورد نیاز و از طرفی این‌که دوره اول رویداد بود، به اندازه کافی کار سنگینی بود که با اضافه شدن اسپانسر و تغییر ناگهانی تم، خرگوش بعدی از راه رسید که خودش دشواری‌های زیادی برای ما به همراه داشت، ولی در انتها این خرگوش را نیز به شکست واداشتیم. خرگوش بعدی، مربوط به سایت رویداد بود که برعکس سایر رویدادها - که از سایت دوره‌های قبل یا سایر رویدادها کمک می‌گرفتند - با توجه به نیازمندی‌های خاص رویداد، بسیار سخت‌تر بود که در این مورد کمی از خرگوش حریف، گل هم دریافت کردیم ولی در نهایت با کمی تأخیر، سایت نیز از راه رسید و می‌توان گفت با این خرگوش نیز تساوی کردیم.

اما این پایان کار خرگوش‌ها نبود. گویا قرار نبود؛ باگربازی را تنها بگذارند. تأخیر در شروع ترم، تعویق در زمان ورود ورودی‌های جدید، تاریخ اردوی ورودی و کلی خرگوش دیگر به دنبال شکست باگربازی بودند.

هم به فکر برگزاری گیم‌جم هستند و خب در این نقطه، تلاقی ما رخ داد و بعد از آن هسته مرکزی اولیه رویداد شکل گرفت. بر خلاف انتظار اولیه من، همه چیز در حال جدی شدن بود. یکی از جالب‌ترین بخش‌های رویداد همان جلسات مجازی بود که آن اوایل سه‌نفری برگزار می‌کردیم و راجع به ساختار رویداد صحبت می‌کردیم. کم‌کم بحث‌های ما جدی‌تر شد و از طرفی مهدی که خودش عضو انجمن علمی بود، پیگیری‌های لازم را در انجمن انجام می‌داد و این شد که رویداد خیلی زودتر از انتظار من به مرحله اجرایی رسید. اگر یک ماه قبل از آن به من - که حتی سابقه استف شدن در رویدادهای دانشکده را نداشتم - می‌گفتید قرار است دبیر یک رویداد باشی، قطعاً باور نمی‌کردم؛ ولی خب آن روز همه چیز واقعی به نظر می‌رسید.

روز صفرم: آغاز فعالیت و چکش‌کاری ساختار رویداد

بعد از کش‌وقوس‌های فراوانی که برای انتخاب اسم رویداد داشتیم، در نهایت «باگربازی» نهایی شد، پیام معرفی رویداد آماده شد، پیام مربوطه حدود نیمه مردادماه روی کانال کافه‌اسسی قرار گرفت و می‌توان گفت از آن نقطه بود که رویداد به صورت رسمی آغاز به کار کرد.

تیم رویداد به سرعت شکل گرفت و از آن نقطه به بعد، چکش‌کاری ساختار رویداد شروع شد. در ابتدا قرار بود رویداد صرفاً شامل بخش حضوری - که مخاطب هدفش بچه‌های دانشکده خودمان بود - باشد، ولی تصمیم‌گیری به این سمت رفت که حیف است در کنار آن یک گیم‌جم مجازی بزرگ‌تر وجود نداشته باشد. این بود که گیم‌جم و به دنبال آن، بخش ارائه‌ها و کارگاه‌ها به رویداد اضافه شدند. رویداد به اندازه کافی بزرگ شده بود و از طرف دانشجویهای مختلف، ایده‌های جذاب دیگری می‌رسید که بعضی از آن‌ها خودشان پتانسیل تبدیل شدن به یک رویداد کامل را داشتند و یکی از دردناک‌ترین بخش‌های رویداد همین کنار گذاشتن ایده‌های جذاب بود، ولی در آن نقطه پذیرفتن بعضی ایده‌های قبلی نیز به اندازه کافی ریسک بزرگی بودند و چاره دیگری نداشتم.

باگربازی

روایتی از مسیر پر از چالش و هیجان اولین دوره رویداد بازی‌سازی باگربازی

دانشگاه صنعتی شریف

۱۳ دقیقه

امیررضا جعفری

کارشناسی ۱۴۰۲



شاید شما هم حس کرده باشید که اگر قرار است رویداد مربوط به بازی‌سازی داشته باشیم، اگر برگزارکننده‌اش قرار نیست دانشکده کامپیوتر باشد، پس کجاست؟ اولین دوره باگربازی بعد از چند سالی که رویداد بازی‌سازی داخل دانشکده و دانشگاه نداشتم، آبان و آذر امسال برگزار شد. ولی نکته اصلی ماجرا این بود که باگربازی با وجود این‌که اولین دوره برگزاری آن بود، آن‌چنان ساده نبود. شامل بخش‌های متعددی بود و داخل بعضی موارد به سمتی حرکت کرده بود که مشابهش داخل هیچ رویداد یا گیم‌جمی وجود نداشت. در این متن علاوه بر صحبت راجع به جزئیات باگربازی قرار است یک روایت درونی‌تر از چالش‌ها و مسیر اولین دوره رویداد نیز ببینیم.

روز منفی یک: چه شد که این ماجرا شروع شد؟

داستان شروع ماجرا خودش به اندازه کافی عجیب است. جرقه اولیه شروع آن را می‌توان به مهرماه سال گذشته و گیم‌کرفت امیرکبیر که خودم شرکت کردم، ربط داد. ولی ترجیح می‌دهم از تیر امسال شروع کنم. جایی‌که دوران بعد از جنگ فرصتی داده بود که با احمد (امیرحسین محمدزاده؛ یکی از دوستان نزدیک من و عضو انجمن علمی دانشکده) صحبت‌های کمی رندوم‌تری داشته باشیم.

احمز از فرم رویداد انجمن علمی برایم گفت و از من خواست پروپوزالی برای این رویداد بنویسم. یک روز بدون این‌که یک درصد احتمال بدهم قضیه قرار است جدی شود، حدود یک ساعت نشستم و از رویدادی نوشتم که شاید بتوان بچه‌های دانشکده را ترغیب کرد برای مدت کوتاهی بازی و بازی‌سازی داشته باشند. بعد از آن احمد از این موضوع برایم گفت که اتفاقاً مهدی علی‌نژاد و امیرمهدی طهماسبی

بازی دانشکده

یکی از ایده‌هایی که در جریان رویداد مطرح شد، بازی دانشکده بود که نسخه اولیه آن به‌عنوان یکی از پایه بازی‌های آماده در اختیار شرکت‌کنندگان رقابت حضوری قرار گرفت. البته تیم علمی بعد از رقابت حضوری، از توسعه آن دست برداشت و در نهایت با طراحی یک مرحله جذاب درون فضای بازی، یک چالش جداگانه برای آن در طول رویداد نیز تعریف کرد. توضیحات تکمیلی راجع به این بازی را می‌توانید در متنی که مهدی علی‌نژاد -هد تیم علمی رویداد- در وبلاگ بایت نوشته‌است، بخوانید.

هجوم خرگوش‌ها (۲)

بعد از سختی‌های رقابت حضوری، گروه دیگری از خرگوش‌ها آماده بودند تا نگذارند حتی برای مدت کوتاهی، تیم باگربازی آب خوش از گلویش پایین برود. بزرگ‌ترین چالش ما برای گیم‌جم، تعداد ثبت‌نامی‌های آن بود. ثبت‌نام گیم‌جم مجازی ما، هم‌زمان با ثبت‌نام رقابت حضوری آغاز شده بود و به‌طور کلی تعداد بخش‌های زیاد رویداد و این مورد که تمرکز ما در ابتدا روی بخش حضوری بود، باعث شده بود مخاطب تا حد زیادی گیج شود و تعداد تیم‌های ثبت‌نامی رقابت غیرحضوری، فقط به اندازه انگشتان یک دست بود. این خرگوش‌ها قوی‌تر از چیزی بودند که در نگاه نخست به نظر می‌رسید. یکی از مهم‌ترین ملاک‌ها و شاید مهم‌ترین ملاک که می‌تواند به افزایش کیفیت و اعتبار یک گیم‌جم کمک کند، قطعاً تعداد شرکت‌کنندگان آن است. ما بسیاری از فرصت‌های مارکتینگ خود را از دست دادیم و با مخاطبی که گیج شده بود، مواجه بودیم.

پوستری با تم جدید طراحی کردیم، سبک و سیاق پیام‌های کانال را عوض کردیم و استقلال گیم‌جم و رقابت حضوری را برای مخاطب تفهیم کردیم. کار ما به همین موارد ختم نمی‌شد؛ مسئول هماهنگی رویداد نیز با کمک تیم مارکتینگ، هر راه نرفته‌ای که در حوزه مارکتینگ وجود داشت را طی کردند تا خرگوش‌ها پیروز ماجرا نباشند. کمک گرفتن از رسانه‌های حوزه بازی، هدف قراردادن کانال‌ها و گروه‌های مربوط به بازی‌سازی و کمپین‌های مارکتینگ جدید از جمله کارهایی بودند که ما برای شکست خرگوش‌ها انجام دادیم. در نهایت، می‌توانم به‌جرت بگویم تیم باگربازی برنده بلامنازع این نبرد در مقابل خرگوش‌ها بود.

کارگاه‌ها و ارائه‌ها

این قسمت، خودش شامل دو بخش بود. سری ویدئوهای ضبط‌شده Godot و ارائه‌های مجازی؛ ویدیوهای ضبط‌شده برای موتور بازی‌سازی Godot به‌نوعی مکمل رقابت حضوری بودند. این ویدیوها توسط تیم علمی رویداد ضبط شدند و از کیفیت بسیار بالایی برخوردار بودند. در بخش بعدی -یعنی ارائه‌های مجازی- مجموعاً ۱۵ ارائه باکیفیت در موضوعات مختلف مربوط به بازی‌سازی و با همکاری متخصصین حوزه بازی‌سازی، برگزار شدند.



رقابت حضوری: تمرکز بر روی دانشکده

این بخش که احتمالاً دلیل اصلی به وجود آمدن رویداد بود، طی دو روز و به‌صورت حضوری درون دانشکده برگزار شد. شرکت‌کننده‌ها در قالب تیم‌های ۲ یا ۳ نفره قرار بود با کمک پایه اولیه بازی‌هایی که توسط تیم علمی رویداد آماده شده بود و در اختیارشان قرار گرفته بود، طی این ۲ روز بازی‌های خود را بسازند و همچنین در فاز آخر رقابت، بازی‌های یک‌دیگر را بازی کنند.

هدف اصلی این بخش، آن بود که تا جای ممکن دانشکده را وارد بازی کنیم و حتی برای مدت کوتاهی کاری کنیم بچه‌های دانشکده بازی و بازی‌سازی را تجربه کنند. اولین قدم این بود که رقابت را به‌نوعی طراحی کنیم که کسانی که قبل از این، تجربه بازی‌سازی نداشتند هم شانس رقابت داشته باشند. برای این مورد، دو کار انجام دادیم. مورد اول، سری ویدیوهای Godot بود که این امکان را فراهم می‌کرد کسانی که تا قبل این تجربه خاصی هم نداشتند، به سطح خوبی از آشنایی با موتور بازی‌سازی Godot برسند. مورد دوم، این بود که سعی کردیم با در اختیار قرار دادن پایه چند بازی آماده، مسیر ساخت اولین بازی آن‌ها را برایشان هموار کنیم.

قدم بعدی این بود که جدیت رقابت را تا جای ممکن کم کنیم و سعی کنیم رقابت شکلی بگیرد که بتواند برای همه جذابیت خود را داشته باشد. اولین کار ما در این مسیر این بود که به جای قرار دادن صرفاً چند جایزه بزرگ، جوایز را به قسمت‌های کوچک‌تر بشکنیم و برای بخش‌های مختلف، جوایز جداگانه قرار دادیم تا شانس همه برای برنده شدن افزایش پیدا کند. قدم دوم این بود که طراحی خود مسابقه را به سمتی ببریم که جذابیتش برای همه افزایش پیدا کند؛ برای این مورد، یک فاز رویداد را کاملاً به بازی کردن اختصاص دادیم و حتی جوایزی مخصوص این فاز نیز قرار دادیم.

با این همه، از آن جایی که دوره اول برگزاری رویداد بود، در بسیاری از موارد، همه چیز مورد انتظار ما نبود و در بعضی از بخش‌ها به جای هجوم خرگوش‌ها با هجوم فیل‌هایی روبه‌رو شدیم که به دنبال فیل کردن رویداد بودند. چالش‌هایی مثل آماده کردن بازی همه تیم‌ها برای فاز آخر، که با تأخیر مواجه شد؛ یا ناامید شدن و کنار کشیدن برخی تیم‌ها در میانه راه، که انتظار آن را نداشتیم. ولی در نهایت، با وجود این که دوره اول رویداد بود، تا حد خوبی این فیل‌ها کنترل شدند و تجربه‌های خوبی برای دوره‌های بعدی (در صورت برگزاری) کسب شدند.



گیم جم مجازی: فراتر از انتظار ما و پیر از چالش

احتمالاً بهتر است ابتدا توصیف کوتاهی از خود این موضوع که ذاتاً گیم جم چیست ارائه کنم؛ «گیم جم یک رویداد فشرده و متمرکز برای ساخت بازی است که تیم‌ها در یک بازه زمانی کوتاه و بر اساس یک تم مشترک، بازی خود را خلق می‌کنند.»

مرحله اول، انتخاب تم بود که بعد از چند جلسه با تیم علمی تصمیم گرفتیم تم را سفر به فضا قرار دهیم. کار دیگری که راجع به تم انجام دادیم این بود که به جای این که صرفاً یک جمله یا کلمه را تم قرار دهیم، یک عکس برای تم در نظر بگیریم تا افراد مختلف بتوانند با برداشت‌های مختلف خود از تم، بازی‌های خود را بدون محدودیت خلق کنند.

مدت زمان گیم جم باگربازی ۱۰ روز تعیین شد و افراد می‌توانستند به صورت انفرادی یا در قالب تیم‌های ۲ تا ۶ نفره در گیم جم شرکت کنند؛ در مورد میزان استقبال از گیم جم نیز با شکست خرگوش‌هایی که در بخش قبل راجع به آن‌ها صحبت کردیم، تعداد ثبت‌نامی‌های گیم جم باگربازی به ۱۲۰ تیم و ۳۵۰ نفر رسید که هم نسبت به رویدادهای مشابه، عدد بسیار بالایی بود و هم تا حدی فراتر از انتظار خود ما بود.

در نهایت نیز، ۷۲ بازی برای داوری ارسال شدند که تعداد خوبی از آن‌ها کیفیت بسیار بالایی داشتند. البته این تعداد بازی خوب نیز خودش خرگوش جدیدی بود که آمده بود کار داوری را برای ما سخت‌تر کند و باعث شده بود چالش جدیدی برای ما ایجاد شود. در این جا نیز ما با چند مرحله‌ای کردن فرایند داوری و افزایش تعداد داوران سعی کردیم دقت داوری را افزایش دهیم و از طرفی تأثیر نظر شخصی را هم تا حد خوبی کاهش دهیم.

مرحله آخر گیم جم، اختتامیه آن بود که حتی آن هم قرار نبود بدون چالش برگزار شود. در ابتدا، آلودگی هوا و تعطیلی دانشگاه باعث شد ما مجبور شویم اختتامیه را به تعویق بیندازیم و بعد از آن، از کار افتادن سیستم صوتی سالن ربیعی، دقیقاً شب قبل از اختتامیه، چالش بعدی ما بود؛ درنهایت، این خرگوش‌ها را نیز به هر نحوی که بود، شکست دادیم و اختتامیه را با کیفیت خوبی برگزار کردیم و از برگزیدگان گیم جم تقدیر کردیم.

روز دوم: تجربه‌ها

برگزاری یک رویداد جدید، از اولش هم مشخص بود قرار نیست خیلی ساده باشد؛ منطقاً این سختی برای باگربازی که خودش شامل بخش‌هایی بود که به طور مستقل می‌توانستند یک رویداد کامل باشند، چند برابر بود. اما نکته مهم میراثی بود که برای دوره‌های بعدی رویداد به جا گذاشتیم و تجربه‌هایی بود که کسب کردیم. دوره‌های بعدی رویداد (در صورت برگزاری) قرار نیست با خیلی از خرگوش‌هایی که ما با آن‌ها جنگیدیم، دوباره بجنگند. با همه مسائل از پیش آمده، با برندی که در دوره اول ساخته شد، قطعاً خرگوشی پیدا کردن اسپانسر یا مسائل مورد نیاز برای مارکتینگ خیلی ضعیف‌تر از خرگوش‌هایی هستند که ما با آن‌ها جنگیدیم.

نیم‌شب قبل باگربازی بعد از دیدن وضعیت سرور SSC خطاب به هد فنی رویداد:



ولی شاید مهم‌تر از این مسائل، تجربه‌هایی بود که از اولین دوره باگربازی کسب کردیم. متوجه شدیم در خیلی موارد قرار نیست برنامه‌ریزی‌ها درست جواب بدهند و همیشه سنجی از آسمان (یا شاید خرگوشی از زیر زمین) می‌تواند نازل شود. فهمیدیم افزایش پیچیدگی بعضی از موارد می‌تواند خیلی گران‌تر از آن چیزی که انتظارش را داریم، تمام شود. دیدیم فرسایشی شدن خیلی از موارد، می‌تواند انگیزه بالانگیزترین اعضای رویداد را هم از آن‌ها بگیرد. ولی همچنین دیدیم که در تاریک‌ترین نقطه‌ها می‌شود راهی برای نجات و بقاء پیدا کرد.

به طور کلی، شاید بتوان از تجربه‌ها و مسائلی که باگربازی در طول مسیر خود با آن‌ها مواجه شد، یک کتاب کامل نوشت. ولی فارغ از همه مسائل، مهم‌ترین نکته احتمالاً این نکته باشد که فارغ از تمام اتفاقات و نگاه‌های مختلفی که از بیرون راجع به شکست یا موفقیت اولین دوره باگربازی می‌تواند وجود داشته باشد، در پشت صحنه، باگربازی تا آخرین لحظه ایستاد و جنگید.

خرگوش را قورت بده

روزی که داشتیم به عنوان دبیر رویداد انتخاب می‌شدم، از آن جایی که تا آن روز حتی استفاده هیچ رویدادی هم نبودم، ترسی برای قبول کردن دبیری داشتم. ولی چیزی که به من چشمک می‌زد، تجربه جذابی بود که حس می‌کردم نباید از دستش بدهم. مسیر دبیری اصلاً ساده نبود و حتی در خیلی از موارد بسیار سخت‌تر از آن چیزی بود که انتظارش را داشتم. در بعضی موارد مجبور شدم از درسم بزنم، کلاس‌هایم را شرکت نکنم و حتی برای رویداد بی‌خوابی بکشم.

ولی باز هم اگر از من بپرسند ارزشش را داشت یا نه، پاسخ قطعاً مثبت خواهد بود. اگر در ۲۰ سالگی قرار نیست این ترس‌ها را کنار گذاشته و در بعضی موارد از چارچوب‌های امن خود خارج شد، پس زمان آن چه موقع است؟ با همه سختی‌ها، قطعاً آورده‌های این مسیر برایم خیلی بیش‌تر بوده است. می‌خواهم حرف‌های کلیشه‌ای -مثل یادگیری و تجربه کار تعاملی و تیمی، که کاملاً هم برایم صدق می‌کنند- را کنار بگذارم؛ حتی اگر تجربه‌های جذابی که در طول مسیر داشتم را هم کنار بگذارم، مهم‌ترین آورده این رویداد، آشنایی من با دوستان جدیدی بود که اگر این رویداد نبود، شاید هیچ‌وقت شانس دوستی و تعامل با هم را پیدا نمی‌کردیم.



ماندن در حاشیه امن و طی کردن روند عادی زندگی، شاید گزینه کم‌ریسک‌تری به نظر برسد، ولی قطعاً بهترین گزینه نیست. دوران دانشجویی احتمالاً بهترین فرصت شما برای انجام کارهای جدید و کسب تجربه‌های جذاب است. پس سعی کنید در بعضی موارد فارغ از ترس از نتیجه کار و حرف دیگران، به دنبال کسب تجربه‌های جذاب باشید. تا زمانی که فرصت آن را دارید و می‌توانید، خرگوش‌تان را قورت بدهید؛ شاید فردا دیگر خرگوشی برای قورت دادن نداشته باشید.

هرچه بگندد نمکش می‌زنند، وای به روزی که بگندد نمک



دانشگاه صنعتی شریف

نیما شیرزادی

۴ دقیقه

کارشناسی ارشد ۱۴۰۱

در دنیای زیرساخت نرم‌افزاری، واژه‌هایی مانند پایداری^۱ و دسترس‌پذیری بالا^۲، دیگر فقط شعارهای مهندسی نیستند؛ بلکه شرط زنده ماندن یک سامانه در محیط‌های واقعی‌اند. همه ما می‌دانیم که اگر بخواهیم سیستم‌هایمان بدون وقفه کار کنند، باید ابزارهایی برای مانیتور کردن آن‌ها داشته باشیم، اما کم‌تر پیش می‌آید کسی از خودش بپرسد: «اگر خود سیستم مانیتورینگ از کار بیفتد، چه خواهد شد؟» این جاست که معنای دقیق آن ضرب‌المثل قدیمی به چشم می‌آید:

«هرچه بگندد نمکش می‌زنند، وای به روزی که بگندد نمک!»

گام نخست: جداسازی زیرساخت مانیتورینگ از زیرساخت اصلی

در طراحی بسیاری از سیستم‌ها، تمام سرویس‌ها، از اپلیکیشن‌ها گرفته تا دیتابیس‌ها و حتی مانیتورینگ در یک بستر مشترک، -مثلاً یک کلاستر کوبرنتیز^۳- مستقر می‌شوند. این کار در ظاهر، ساده، منسجم و مقرون به صرفه است؛ اما در عمل، یک نقطه تکی شکست^۴ پنهان در دل این معماری وجود دارد. اگر به هر دلیل، زیرساخت اصلی دچار اختلال شود، مثلاً گره مستر از دسترس خارج شود یا حافظه مرکزی آسیب ببیند، در همان لحظه سیستم مانیتورینگ هم همراه با کل زیرساخت سقوط خواهد کرد. نتیجه روشن است: وقتی همه چیز خراب می‌شود، دیگر هیچ چیز نیست که خرابی را به ما گزارش دهد. به همین دلیل، نخستین تصمیم حرفه‌ای در معماری زیرساخت، آن است که استک مانیتورینگ را به‌طور کامل از کلاستر اصلی جدا کنیم. این استک باید در محیطی مجزا و خارج از منابع و شبکه‌ای که زیرساخت اصلی روی آن قرار دارد، اجرا شود. حتی بهتر است در یک دیتاسنتر یا منطقه فیزیکی متفاوت مستقر شود تا در صورت بروز حادثه،

همچنان سالم باقی بماند. با این جداسازی، اگر زیرساخت اصلی از کار بیفتد، سامانه مانیتورینگ همچنان می‌تواند از بیرون وضعیت آن را بسنجد و هشدارهای لازم را ارسال کند.

اما همین جا متوقف نمی‌شویم؛ مانیتورینگ هم باید مانیتور شود

بسیاری از تیم‌ها وقتی استک مانیتورینگ را از زیرساخت اصلی جدا می‌کنند، احساس امنیت می‌کنند و تصورشان این است که دیگر نیازی به نگرانی نیست. اما واقعیت این است که حتی این استک مستقل هم می‌تواند دچار خرابی شود. ممکن است سرویس پرومیتوس^۵ قفل کند، مدیریت‌کننده هشدار^۶ از ارسال هشدار بازماند یا شبکه‌ای که گرافانا^۷ روی آن قرار دارد، به‌طور موقت قطع شود. در چنین شرایطی، ما باز هم همان مشکل قدیمی را داریم: «دیده‌بان خوابش برده است». راه حل حرفه‌ای برای این وضعیت، طراحی یک لایه دوم مانیتورینگ است؛ سامانه‌ای کوچک‌تر و ساده‌تر که در نقطه‌ای متفاوت از شبکه یا حتی در یک کشور دیگر قرار دارد و فقط یک وظیفه دارد: بررسی سلامت استک مانیتورینگ اصلی.

این لایه ثانویه ممکن است چیزی به سادگی یک Blackbox Exporter، یک VPS با اسکریپت‌های ping و health check یا حتی یک سرویس خارجی مانند UptimeRobot باشد که فقط بررسی می‌کند آیا اندپوینت‌های حیاتی پرومیتوس، گرافانا یا مدیریت‌کننده هشدار در دسترس هستند یا نه. اگر این سرویس‌ها پاسخ‌گو نباشند، این سامانه کوچک باید بتواند از مسیرهای جایگزین مثل پیامک، تماس تلفنی یا ایمیل از سرور خارجی هشدار ارسال کند.

به بیان دیگر، سیستم مانیتورینگ هم باید مانیتور شود.

نیما در پایان این شماره
بالاخره بابایت آشتی کرد!



این همان اصل ساده‌ای است که در بسیاری از طراحی‌ها نادیده گرفته می‌شود؛ اما در لحظه بحران، مرز بین آگاهی و نابینایی کامل است.

معماری پیشنهادی برای مانیتورینگ چندلایه

یک طراحی بهینه برای مانیتورینگ پایدار می‌تواند سه لایه داشته باشد.

- ۱. زیرساخت اصلی:** شامل سرویس‌ها، اپلیکیشن‌ها و دیتابیس‌ها که منبع اصلی داده هستند.
- ۲. استک مانیتورینگ مستقل:** مجموعه ابزارهایی مانند پرومیتوس، گرافانا یا الرت‌منیجر؛ که بیرون از این زیرساخت مستقر شده‌اند و وضعیت آن را پایش می‌کنند.
- ۳. سیستم مانیتورینگ لایه دوم:** سامانه‌ای کوچک و جدا که وظیفه‌اش بررسی سلامت خود استک مانیتورینگ است.

این سه لایه بودن باعث می‌شود حتی در بدترین سناریوها، همیشه یک نقطه دید خارجی باقی بماند که بتواند تشخیص دهد چه چیزی از کار افتاده و تیم فنی را مطلع سازد.

در نتیجه، اگر از دید فلسفی به سیستم‌ها نگاه کنیم، مانیتورینگ در حقیقت ضمیر آگاه زیرساخت است. این ضمیر، همیشه باید بیدار بماند، اما هیچ ضمیری بدون مراقب، مصون از خواب نیست. وقتی ما استک مانیتورینگ را از زیرساخت اصلی جدا می‌کنیم، در واقع گام نخست را برای مصون کردن آگاهی از مرگ برداشته‌ایم. اما گام دوم قرار دادن مراقبی برای خود مانیتورینگ است که پایایی واقعی را تضمین می‌کند.

به زبان ساده، در دنیای فناوری هم مانند زندگی، همیشه باید به این فکر کرد که اگر روزی «نمک» بگندد چه باید کرد.

سیستم مانیتورینگ ما همان نمک است؛ حافظ سلامت بقیه. و اگر خودش از کار بیفتد، دیگر هیچ سازوکاری برای فهمیدن خرابی‌ها باقی نمی‌ماند. پس بیاید همان قدر که برای نظارت بر دیگران تلاش می‌کنیم، برای نظارت بر «ناظر» نیز تدبیر ببندیم، چرا که در مهندسی زیرساخت، هیچ چیز خطرناک‌تر از دیده‌بانی نیست که کسی حواسش به او نیست.

100 Magazine

این نشریه بیش از آن که حاصل جمع چند متن باشد، جهانهای بود برای نزدیکتر شدن دل دانشجویان هم دغدغهای که دغدغههای مشترک، ذهنشان را به هم گره زده بود. شاید در میان مشغولیت‌هایمان، گاهی وقت‌مان را به «با هم بودن» اختصاص دادیم و کوشیدیم از «بایت» پلی بسازیم برای پیوند، گفت‌وگو و هم‌دلی. خواستیم ذهن‌ها، ایده‌ها و عقیده‌هایمان را با دیگران به اشتراک بگذاریم، از نقد نهراسیم و پذیرای دیدگاه‌های متفاوت باشیم. اگر «بایت» توانسته باشد حتی گامی کوچک در نزدیکی بیشتر دانشجویان دانشکده مهندسی کامپیوتر و پویایی فضای علمی و فرهنگی آن بردارد، همین برای ما مایه افتخار است.

انتقادات و پیشنهادات خود را
با مادر میان بگذارید



SSC
STUDENTS' SCIENTIFIC CHAPTER
انجمن علمی دانشکده مهندسی کامپیوتر
دانشگاه صنعتی شریف