Jay Blanchard — Narrative Résumé 0.1

Senior Technology Leader | Platform Engineering | Developer Experience | Cloud Infrastructure | AI Enablement

> **I got tired of pretending a 30-year engineering career is best explained as a chronological collection of bullet points.**

So this résumé starts somewhere else: with the problems I have spent my career solving.

The technologies have changed enormously. The pattern hasn't. I tend to arrive where something important works, but the machinery around it is making the work harder than it needs to be. Delivery is fragile. Infrastructure has outgrown itself. Teams don't understand one another. Knowledge lives in people's heads. A business needs to make a significant change without interrupting the business it already has.

My job has usually been to understand the whole system—technology, people, process, and business—and help make the next version possible.

Change the Machinery While It's Still Running

Replacing a system is relatively easy when you can turn the old one off.

Most of my career hasn't afforded that luxury.

At **Pocket Communications**, a third-party billing platform was approaching a **$1 million renewal**. There was nothing fundamentally wrong with it. The business simply asked whether we could build our own.

I knew the system and its APIs after four years of working with it. I also knew the important question wasn't whether my team *could* build a replacement. It was whether we could build one while the existing system continued serving the business.

I mapped the work, dependencies, people, and timing. My plan said **90 days to build it and another 10 to pressure-test it**. We assembled a deliberately small cross-functional team: three software engineers, fractional systems and database administrators, a designer, and me serving wherever necessary—program manager, architect, developer, cat-herder, and occasionally shrink.

We built the replacement alongside production, cloned and isolated the databases, practiced the data delta, and wrote the cutover playbook.

Our planned Day 95 cutover failed its data checks.

So we rolled back.

On Day 99, we tried again. The cutover went so smoothly that we wondered whether anything had happened.

Something had: roughly **450,000 customers** had moved to the new billing platform without knowing anything had changed, and Pocket avoided the $1 million renewal.

Years later at **Constant Contact**, the scale was different but the principle was the same. Payments-as-a-Service—roughly nine applications—needed to move from on-premises infrastructure to AWS. We started with the simplest application, built and validated through DEV/STAGE/PROD environments, progressively migrated workloads and data, and deliberately left the largest internal application until last.

The first major cutover failed because of networking issues.

We rolled back.

A few days later, we did it successfully.

Good migrations aren't defined by never failing. They're defined by making failure recoverable.

Understand the Work. Remove the Friction.

Long before anyone called it Developer Experience, I was trying to make technology disappear from the parts of people's jobs where it didn't belong.

At Pocket, we even named internal systems **EZ This** and **EZ That**. The owner once joked that I'd love a giant red button I could push every morning that simply made everything work.

He wasn't entirely wrong.

We built systems connecting third-party dealers directly with billing and provisioning. We automated financial processes. We integrated call-center systems and created operational dashboards. We built and patented an ACH process that allowed dealer payments to post during the day while moving funds automatically each night.

But one of the most important things I did required almost no technology.

I put developers and database engineers beside people doing the business work.

Once engineers understood *why* something mattered, requirements got better. Engineers proposed solutions the business hadn't thought to request. They pushed back differently. Business people learned how to ask engineering for what they needed.

At **Fugro**, the same principle applied to offshore survey technology. Developers learned what surveyors actually encountered aboard vessels. Surveyors learned how the engineering process worked. Software engineers became better at explaining requirements to the electrical engineers building the IoT hardware.

At **Newfold Digital**, the goal became a paved delivery road: Jenkins running on Kubernetes via Helm, infrastructure and agents defined as code, ephemeral Docker-based agents running as Kubernetes pods, reusable pipeline templates, semantic versioning, Artifactory, AWX, and delivery patterns capable of supporting OCI.

But the objective was never to make engineers experts in Jenkins.

Quite the opposite.

Engineers should be able to focus on engineering. QA should be able to focus on testing. Delivery machinery should help rather than demand attention.

Removing friction, however, doesn't mean removing understanding. Application teams still needed to own their pipeline templates, understand how their applications moved from bare metal to containers, and learn enough about the delivery system to make mistakes safely and operate what they built.

Automate the work that teaches nothing.

Don't automate away ownership.

Learn. Do. Teach. Trust.

Some of my earliest leadership lessons came before my technology career.

At the Air Force military working-dog veterinary center, leadership deliberately rotated us through the entire mission: surgery, dental, necropsy, imaging, daily care, purchasing—even taking care of the grounds.

Rank wasn't an excuse for not understanding the work.

That lesson followed me.

When I arrived at Fugro, OARS already existed aboard vessels, but development and deployment were largely siloed. Code spanned Haskell, C#, C++, and the web front end.

Releases to on-premises systems and IoT devices depended heavily on Make. There was little meaningful version-control discipline and no mature release process.

I took ownership of Docker while working side-by-side with an AWS partner on the cloud architecture and release model. We incorporated containerization and versioning into the build process, introduced Jenkins pipelines, moved engineering into Git, and progressively taught the OARS engineers to own the system themselves.

By the time I left, the team didn't need me to explain how their delivery system worked.

That was the point.

At Constant Contact, teaching expanded beyond a single team. Together with engineers from two other groups, I developed a five-part Docker education series with presentations, examples, and repositories.

More than **100 engineers** showed up for the first lunch-and-learn.

We broke the Wi-Fi.

I later received a **Rovie Award for Engineering Excellence** for helping evangelize DevOps practices across the organization.

Teaching isn't something I do after the engineering is finished.

Teaching is part of the engineering.

Start With the Problem, Not the Technology

My first questions are usually:

What is your goal?

Show me what you're talking about.

And then:

Don't answer right away.

At **Apex Innovations**, I inherited a small healthcare continuing-education business whose technology had grown organically alongside its success. Individual courses had effectively become bespoke applications. Code lived on local servers. Different technologies had accumulated over time. Medical professionals, a graphic designer, developers, and leadership all contributed to course production, but versioning and approval were loose.

Certification requirements meant the company already understood why provenance mattered. I used that common ground to introduce Git-based version control, pull-request visibility, controlled graphics and content, and a more structured—initially very manual—versioning process.

As we unified the applications, another opportunity became obvious:

Stop rebuilding the machinery for every course.

Templates, repeatable structures, and shared capabilities made both existing-course updates and new-course production substantially more efficient.

Success then exposed another constraint: servers in a closet.

I researched the alternatives, initially favoring Rackspace largely on cost. Further investigation showed that AWS offered capabilities Apex needed that Rackspace didn't provide at the time.

So the answer changed.

We chose the relatively lightweight **NIH Stroke Scale course** as our production proving ground. Its existing version could continue operating on-premises while we built and tested the AWS version with basic compute and MySQL infrastructure. Cutover largely became a matter of redirecting the access point.

The experiment showed faster load times, particularly valuable for graphics-heavy material; better image management through S3; improved routing through Route 53; database capability we couldn't economically reproduce locally; less on-premises administration; and better speed to market.

When I left, we were preparing the major courses for migration and continuing to bring inherited systems into the light.

The cloud wasn't the strategy.

Removing the constraints on the business was the strategy.

Make the Problem Legible

When Constant Contact was separated from what became **Newfold Digital**, the requirement sounded deceptively simple:

Newfold would receive the existing Payments-as-a-Service platform. Constant Contact still needed to operate independently with effectively the same platform—but only its own data.

The TSA program was floundering.

I went looking for the answer.

Years of infrastructure-as-code, Git discipline, automated delivery, AWS architecture, documentation, and PCI work meant much of what we needed was already reproducible.

So I decomposed the problem application by application:

What must be duplicated?

What can remain temporarily shared?

Whose data belongs where?

Which dependencies have to be broken?

In what order?

Who owns each piece?

What proves that separation is complete?

The largest risk wasn't application infrastructure. It was the database: live payment data, stretched database teams on both sides, replication, eventual separation and purging, and PCI controls that could never be compromised.

I converted the ambiguity into a timeline the PMO and executive leadership could operate against, identified when specific teams were needed, and made explicit where leadership intervention would be required for contractor or cross-company decisions.

When Constant Contact's contractors couldn't develop the necessary capability quickly enough despite extensive side-by-side training and documentation, my manager asked a simple question:

“Can't you just do all of this?”

I could.

I relied on the database and network teams for their specialties and took ownership of the remaining gap: CloudFormation, pipelines, scripts, tokenization operations, application work, documentation—the whole nine or ten yards.

The lesson wasn't that one person should do everything.

It was that somebody needed to understand enough of the whole system to recognize **what wasn't owned** and make sure it became owned.

Know Where the Grease Is

I have watched technical leadership become less effective as its distance from the work increases.

A leader doesn't have to write the code.

A leader does need to understand **how much grease is involved**.

That proximity lets me know when *not* to intervene.

On Newfold's Developer Platform, I repeatedly had to remind myself that application teams needed to complete their own pipeline templates. They needed to understand their own containerization. They needed room to make recoverable mistakes.

I could have automated much more of the work myself.

Doing so would have optimized delivery at the expense of ownership.

But proximity also tells me when to step in. During the Constant Contact TSA, teaching and documentation eventually stopped closing the capability gap quickly enough to protect the deadline. At that point, leadership meant putting my own hands back on the machinery.

And sometimes proximity means standing **in front of the team**, rather than beside it.

During Pocket's billing replacement, I went head-to-head with the CIO more than once so developers and database engineers could concentrate on the work rather than defending its complexity upward.

A leader close to the work can tell the difference between an excuse and an engineering reality.

And can defend the latter.

Build the People Who Build the Systems

I've interviewed hundreds of engineers.

What am I looking for?

Aptitude and ease.

Aptitude is the ability to think, learn, connect ideas, and work through something unfamiliar.

Ease isn't charisma. It's knowing what you're talking about, being honest about what you don't know, and allowing an interview to become a collaborative technical conversation instead of an adversarial performance.

Arrogance can erase both.

At Pocket, an extremely quiet employee from one of our retail stores approached me after a company event and asked whether I needed software engineers. She had recently completed an associate degree in computer science.

I invited her to interview the following week.

No coding test.

We talked technology.

We talked **why**.

We talked **how**.

There was a spark.

I called her manager immediately afterward, walked to HR, got approval from a VP for the money, and brought her back the next day to offer her the job.

She jumped into engineering before I had any opportunity to throw her in. The experienced developers and DBAs embraced her. She held her own with all of them.

I once walked into our skunk works and found her working simultaneously on two keyboards with two separate trains of thought.

Sometimes the strongest engineer in the room hasn't become an engineer yet.

But sparks aren't the only people who make engineering organizations work.

There are also the steady engineers.

They're the Sun.

They're dependable. They carry enormous tribal knowledge. They remember why systems are the way they are. I deliberately bounce ideas off them publicly because other people need to see the value of that steadiness.

Good leadership isn't assembling a team of people who look alike.

It's seeing the person who's actually in front of you.

Bet on aptitude. Teach judgment. Expect humility. Protect curiosity. Recognize steadiness.

Technical Landscape

Cloud & Infrastructure
AWS • OCI • Kubernetes • Docker • Helm • CloudFormation • Infrastructure as Code • S3 • Route 53 • ECS • IAM • Load Balancing • Networking • Disaster Recovery

Platform & Delivery Engineering
Jenkins • CI/CD • Git • Bitbucket • GitHub • Artifactory • AWX/Ansible • Maven • Semantic Versioning • Pipeline Templates • Ephemeral Build Agents • Developer Experience • Paved Roads

Software & Architecture
REST APIs • Distributed Systems • Java • PHP • C#/.NET exposure • C/C++ • SQL • MySQL • SQL Server • Tomcat • Application Modernization • Systems Integration

Security & Governance
PCI-DSS • Secure Payments • Architecture Reviews • Vulnerability Remediation • Penetration Testing • GDPR • IAM • Audit Support • Tokenization

Leadership
Platform Strategy • Engineering Leadership • Technical Program Leadership • Cloud Migration • M&A/TSA Separation • Developer Enablement • Hiring & Coaching • Cross-functional Leadership • Vendor/Partner Management • Technical Education

---

Employment History

The stories above span organizations and roles including:

Newfold Digital / Constant Contact
Senior Principal Software Software Engineer
FEBRUARY 2019 - JANUARY 2026

Principal-level platform and developer-experience engineering; Jenkins/Kubernetes platform modernization; OCI delivery enablement; large-scale application migration; AI-assisted pipeline diagnostics. Payments-as-a-Service cloud migration; AWS delivery architecture; PCI

engineering and audit responsibility; DevOps/Docker education; engineering excellence recognition; separation/TSA leadership.

Fugro — OARS (Operator Assisted Remote Services)
Systems Engineer - DevOps / CICD Team Lead
APRIL 2014 - FEBRUARY 2019

OARS offshore survey platform; Docker/AWS/Jenkins modernization; Git and release engineering; global delivery architecture; engineering and surveyor enablement.

Apex Innovations
Director of Technology
NOVEMBER 2012 - DECEMBER 2013

Healthcare continuing-education technology; application/platform modernization; Git/version governance; AWS adoption and migration preparation.

Pocket Communications
Director of Information Technology
FEBRUARY 2006 - AUGUST 2011

Director-level engineering and technology leadership; billing and provisioning; payments/ACH; call-center and finance integrations; operational automation; custom billing replacement; hiring and team development.

Earlier Career — Software engineering, systems integration, e-commerce, and technology modernization across additional organizations and industries.

What I Bring

I don't have one favorite cloud, framework, programming language, or engineering methodology.

I have favorite **outcomes**.

Systems people understand.

Delivery people trust.

Infrastructure that can be reproduced.

Engineers who own their work.

Technology that makes the business easier rather than becoming another part of the business that needs to be managed.

And organizations capable of changing important machinery **without turning everything off first.**

That's the work I want to keep doing.