FROM FUNDAMENTALS TO MASTERY

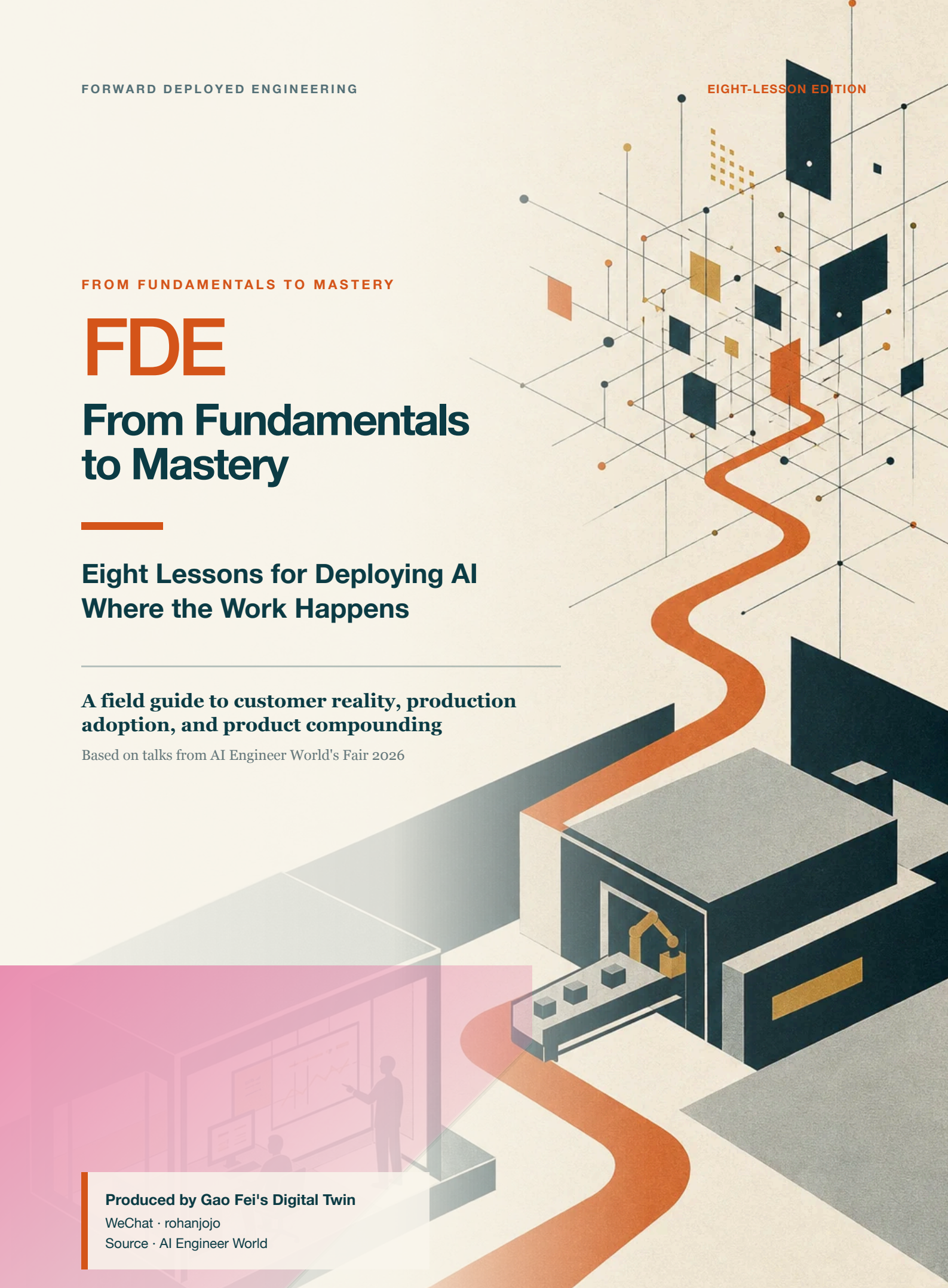
FDE

From Fundamentals to Mastery

Eight Lessons for Deploying AI Where the Work Happens

A field guide to customer reality, production adoption, and product compounding

Based on talks from AI Engineer World's Fair 2026



Produced by Gao Fei's Digital Twin

WeChat · rohanjojo

Source · AI Engineer World

CONTENTS

Inside the book

From the origin of the role to a practical operating system for production AI.

Preface: When Code Gets Cheap, Reality Becomes the Bottleneck

The modern AI demo has become almost suspiciously easy to make. A model writes the function. An agent calls an API. A polished interface appears before lunch. In a controlled environment, the result can look like a finished product.

Then the customer asks for something that sounds minor.

Can it use the permissions we already have? Can it write the result back into SAP? What happens when the account belongs to two billing entities? Can the security team inspect every action? Who owns the failure when the agent sends the wrong refund? Can we run it inside our cloud? Can our operations team change the policy without waiting for your engineers?

None of these questions is a rejection of AI. They are what it means to use AI inside a real organization.

Enterprise work is held together by old systems, local vocabulary, unofficial spreadsheets, exception paths, approval habits, and judgments that no one has written down. A request that arrives as “build an agent for finance” may conceal six different jobs: discovering what people actually do, deciding which part is worth changing, connecting the necessary systems, building a safe control surface, measuring whether the workflow improved, and turning what was learned into a reusable product capability.

Forward deployed engineering sits inside that gap.

The name came to prominence through Palantir, where engineers worked close to customers and often inside their technical environments. In 2026, the label appears across model companies, agent startups, enterprise software vendors, and internal AI teams. Yet the jobs behind it are far from identical. Some FDEs deploy infrastructure. Some integrate data. Some behave like product managers with production access. Some design agents, write application code, train customer teams, or rebuild a workflow from first principles.

That ambiguity is not a reason to dismiss the field. It is a clue. The boundary between a software product and the organization using it has become the most important engineering surface in enterprise AI.

This book develops that argument through eight talks from the Forward Deployed Engineering track at AI Engineer World's Fair 2026. The speakers come from Sierra, Anthropic, Kepler, Varick Agents, Ramp, Decagon, Cognition, and Factory. Each describes the work from a different position: the history of the role, the economics that make it viable, the art of problem discovery, the mapping of hidden workflows, the discipline of scoping, the path from customization to self-service, the customer environment as an evaluation system, and the verification loops behind autonomous software development.

The chapters are not transcripts. Spoken repetition has been removed, examples have been restored to their argument, and unfamiliar products or practices are explained at first use. The original talks remain the primary source. Public company documentation is used to verify names and supply context, not to overwrite what the speakers said. Numbers drawn from talks should be read as speaker or company claims unless an independent source is explicitly identified.

The result is meant to be useful in three ways.

For a founder, it is a test of whether an FDE team is actually needed—and whether the company has enough platform underneath it to avoid becoming a custom development shop. For a product or engineering leader, it is a field guide to turning customer-specific work into reusable capability. For an engineer, it is a description of a job in which technical ability matters enormously, but judgment, curiosity, and accountability often matter more.

The central question is simple:

> When an AI system leaves the demo and enters the place where work really happens, who takes responsibility for making the whole loop work?

The eight lessons that follow are eight answers to that question.

The Gap FDE Is Designed to Close

Imagine a company that sells an AI agent for accounts receivable. The sales conversation begins with a clear promise: the agent will reduce the time employees spend chasing overdue invoices. The model can draft emails, summarize customer history, and choose a follow-up action. The prototype is convincing.

Production begins with a less glamorous list. Customer identity is represented differently in the CRM and the ERP. Payment status arrives late from a banking feed. Some accounts must never receive automated messages. A regional team uses a spreadsheet to override credit policy. The legal team needs an auditable explanation for every action. The collections manager judges success by cash recovered, while the project sponsor is reporting hours saved. The “simple” agent is now a system that crosses data, software, policy, and organizational authority.

A conventional software company divides this work among sales, solutions architecture, implementation, product, engineering, support, and customer success. Specialization is sensible, but it creates a failure mode: every team can finish its own task while the customer still receives no working outcome. The sales team captured the request. The product supports the feature. The integration was delivered. The model met its benchmark. The users nevertheless avoid it because it interrupts the way exceptions are handled.

FDE is an organizational answer to this fragmentation. It keeps someone close enough to the entire path—from problem discovery through production adoption—to see where the result is being lost.

What the Customer Is Actually Buying

Enterprise buyers rarely wake up wanting an ontology, an agent harness, or a new vector index. They want fewer support escalations, faster close, lower incident time, more reliable planning, or a new service that was previously too expensive to operate. The technology is an implementation detail until it changes one of those outcomes.

That creates a translation problem. The buyer speaks in outcomes. Existing systems expose tables, APIs, permissions, and failure modes. The AI system works with prompts, tools, context windows, traces, and evaluations. The user lives inside a workflow with interruptions and exceptions. Someone has to make these descriptions refer to the same thing.

The FDE does not merely translate words between groups. The deeper task is to convert an aspiration into a closed, testable loop:

1. A real signal enters the system.
2. The system gathers the right context.
3. An agent or human makes a decision.
4. An action reaches the system of record.

5. The outcome becomes visible.
6. Failure generates evidence that improves the next run.

If any link is missing, the deployment may still produce a good demonstration. It has not yet produced a reliable operation.

Five Layers of the Job

The eight speakers use different titles and organizational models, but their work repeatedly falls into five layers.

1. Reality discovery. Observe how the work is actually done. Identify the people, systems, constraints, exceptions, incentives, and informal workarounds that a requirements document omits.

2. Problem judgment. Decide what should be built now, what should be deferred, and what should not be automated. Narrow a broad ambition into a first outcome that is valuable, safe, and measurable.

3. Production engineering. Connect data and tools, design permissions and human controls, build the agent or application, instrument it, and make it survive the customer environment.

4. Product learning. Separate customer-specific residue from patterns that should become platform primitives, connectors, templates, evaluations, or product features.

5. Organizational change. Help users adopt a different way of working. Transfer knowledge, define ownership, design escalation, and ensure that the new system can operate without the original FDE in the room.

No single person has to perform every layer forever. In fact, a team that depends on one heroic generalist is fragile. But every layer needs a clear owner, and the connections between them must survive handoffs.

A Quick Test: Do You Need an FDE Team?

The title is fashionable enough that companies can adopt it before they understand the function. A useful initial test is to ask four questions.

- Is the product technically powerful but difficult for the buyer to configure or extend?
- Does value depend on the customer's private data, systems, policy, or workflow?
- Do customers differ enough that a fixed implementation repeatedly falls short?
- Can customer-specific work be assembled from a shared platform rather than built from scratch each time?

The first three conditions create demand for close technical deployment. The fourth determines whether the model can become a scalable software business.

If the product is technical and the users are engineers, strong documentation, developer relations, and solutions architecture may be enough. If the product is standardized and the buyer is nontechnical,

conventional implementation and customer success may work better. If every engagement requires an independent codebase, the company may be doing valuable consulting, but it is not getting the economics of a platform-based FDE model.

How to Read the Eight Lessons

The lessons move from role design to field practice and finally to autonomous systems.

- **Lessons 1–2** explain where FDE came from and when the model makes economic sense.
- **Lessons 3–4** show how the real problem is discovered inside messy work.
- **Lessons 5–6** examine scoping, restraint, specialization, and productization.
- **Lessons 7–8** treat the customer environment as an evaluation system and ask how verification changes the ceiling of agent autonomy.

Together they describe a loop: enter the field, find the real problem, define a bounded outcome, build it into production, learn from the failures, and return the reusable part to the product. The quality of an FDE organization is not measured by how many emergencies it can heroically absorb. It is measured by whether every deployment makes the next deployment better.

Lesson 1 — The Role That "Doesn't Exist"

Natalie Meurer, Sierra · *The Dirty Secret of Forward Deployed Engineering*

This lesson: how FDE accumulated four generations of responsibility; why the old jobs never fully disappeared; and why continuous accountability matters more than the title.

Natalie Meurer began with a provocation: forward deployed engineering does not exist.

She did not mean that the people are imaginary. She meant that the label has been stretched across so many kinds of work that it no longer identifies a single profession. One FDE may keep an on-premise installation alive. Another may integrate data and model an ontology. A third designs an application for a known business problem. A fourth teaches the customer to build on a platform. At an AI company, the same title may cover prompt design, tool integration, evaluation, workflow redesign, and production operations.

The common element is not a technical specialty. It is proximity and accountability to a customer outcome.

Meurer arrived at that conclusion through experience. After studying technology policy at Georgetown and learning to code, she joined Palantir in 2016. She worked across privacy, infrastructure, law-enforcement, and defense deployments. She later went to Stanford Graduate School of Business and joined Sierra, where she led agent engineering. Sierra builds AI agents for customer service—systems that converse with customers across channels and take actions in business systems. The work sits precisely where software behavior, company policy, and measurable customer outcomes meet.

When Meurer joined Sierra's deployment team, she disliked the name. The group was not simply installing a finished package. It was taking responsibility for a changing AI system inside a changing customer workflow. In 2024 she proposed "agent engineer" as a better description: a subdiscipline of AI engineering with the customer accountability of forward deployment. Soon the market produced still more labels, including harness engineering—the work of surrounding a model with context, tools, permissions, memory, validation, and runtime controls.

The expanding vocabulary did not solve the boundary problem. It exposed it. The work itself keeps moving as the technology underneath it matures.

Four vintages of FDE

Meurer explained that movement through four periods in Palantir's history. The dates are best understood as dominant bottlenecks, not as clean corporate eras. Each new layer arrived because the previous layer began to work. Crucially, the older responsibility did not vanish.

2008: keep the platform standing

Early Palantir deployments were often installed in customer-controlled environments, including secure networks and on-premise infrastructure. “Forward deployed” was close to literal: engineers worked where the software ran because the environment could not be understood—or sometimes even reached—from headquarters.

The job resembled a mixture of DevOps, site reliability engineering, and field support. DevOps is the practice of making software development and operations jointly responsible for building, releasing, observing, and recovering a system. Site reliability engineering applies software methods to availability and operational risk. Early FDEs carried both instincts into environments that were not standardized and could not simply be handed to a centralized cloud operations team.

Meurer's own Palantir onboarding included deploying the software onto an Amazon EC2 instance, a virtual server in Amazon Web Services. The exercise captured the original constraint. If the platform could not be installed and kept alive, everything above it was theoretical. A customer email about a machine being unplugged at 2 a.m. could become an engineering emergency. The problem was not sophisticated, but the customer's work had stopped.

This period established a durable rule: engineering responsibility did not end when code was merged or software was delivered. If the system did not work where the customer needed it, the job was not complete.

2012: make the data usable

Once the platform became more stable, the next failure became obvious. Data integration software without integrated data is, in Meurer's analogy, a movie theater showing no movie. The building may be impressive, but no one has a reason to sit down.

Customer data lived in many systems, used inconsistent identifiers, inherited years of quality problems, and came with complicated permissions. Moving twenty databases into one location was not enough. A business user should not need to understand table names and joins to ask which aircraft is grounded, which supplier is late, or which account is exposed.

Palantir's answer was the **ontology**: a shared operational model of the business. Rather than presenting raw tables, the platform organizes data as recognizable objects, relationships, and actions—aircraft, parts, orders, organizations, and the things an authorized user can do with them. An ontology is more than a static data dictionary. It is the layer in which data, application logic, permissions, and operational action refer to the same business concepts.

FDEs were well placed to build it because they could work at both ends. They could inspect schemas and pipelines, then sit with users and ask what a field meant in practice. The role now included platform operations **and** data integration **and** business modeling.

2016: turn data into decisions

Usable data created a new question: what should people do with it?

Palantir FDEs increasingly built custom applications and dashboards. Meurer recalled Slate, a drag-and-drop application builder that connected interface components to integrated data. A tool of that kind could quickly turn a modeled dataset into an investigation screen, operational dashboard, or decision workflow. Because the

FDE understood both the customer and the data, it was natural for the same person to assemble the first solution.

But visibility alone was not the outcome. A dashboard that only reads data and cannot write an action back to the relevant system tends to decay. Users see an exception, switch to email or another application to resolve it, and leave the dashboard's state behind. The display becomes separated from the work.

The more important move was from **data to decision**: connect observation, judgment, action, and updated state. This is where the field project became product discovery. Customers rarely arrived with a complete application specification. They wanted a faster or safer decision. The FDE had to infer what interface and workflow would make that possible, then determine which parts belonged to one customer and which were evidence for the product roadmap.

That power also created custom-solution debt. A fast field application could become something the customer used every day. It had not necessarily been designed or tested like a core product, but it could no longer be casually removed. “Temporary” work had entered production.

2020: enable the customer to build

By the time Palantir approached its public listing, sending an engineer to every problem in Finland, Canberra, or a secure facility was not a sufficient scaling strategy. Foundry had matured into a platform, and part of the FDE's work moved toward enablement.

Foundry is not a single database. It is Palantir's data operations platform, combining data integration, the ontology, analytics, applications, workflows, security, and governance. The richer the platform became, the more customers and partners could build for themselves—if someone helped them learn the system and translated initial needs into reusable patterns.

Meurer used Airbus Skywise as an example. Skywise, launched by Airbus with Palantir, brings together data about aircraft, fleets, maintenance, parts, and operations. A small group of Palantir engineers could not permanently build every application needed by thousands of Airbus engineers and aviation users. The larger impact came from enabling those users to create on top of the platform.

The same idea appears in AIP Bootcamps. AIP is Palantir's platform for connecting generative AI to enterprise data, logic, and operations. In a Bootcamp, business and technical users work with Palantir teams to turn a real use case into a working system in a concentrated period. The FDE is no longer only the builder. The FDE teaches, supplies patterns, and helps the customer's own capability spread.

In 2026 Palantir also documented AI FDE, an agentic capability that can perform data transformation, repository work, and ontology changes through natural-language interaction. The loop is revealing: the platform begins to absorb work once done manually by field engineers, while customers use the new capability to attempt more ambitious operations. Automation moves the role upward rather than making it disappear.

The old work stays underneath

These periods did not replace one another. They stacked.

The application still depends on integrated data. The data still depends on a stable platform. Customer self-service still needs sound primitives, permissions, and operational support. An FDE hired in 2026 may be expected to understand all four layers while also behaving like a senior engineer, solutions architect, teacher, product manager, and commercial partner.

Meurer called these differences **FDE vintages**. Someone formed in the infrastructure era may excel at deployment and recovery under pressure. A data-era FDE may be strongest at schemas, permissions, and ontologies. A solutions-era FDE may have sharper product intuition. An enablement-era FDE may be unusually good at abstraction, training, and organizational spread. Asking “what vintage are you?” can reveal more than asking how many years someone has held the title.

It also helps explain why Palantir alumni have founded so many enterprise software companies. Field work exposes people to infrastructure, data, product, customer behavior, commercial constraints, and organizational change in rapid succession. It is an unusually intense school for generalists who can form a useful model of an unfamiliar environment.

Preserve continuity, not heroics

If the title is incoherent, what should survive? Meurer's answer is **continuity of customer accountability**.

Someone—or a tightly connected team—must retain the thread between why the customer bought, how the system was designed, why users adopt or reject it, what failure costs, and what the product team should learn. Without that continuity, sales, implementation, product, engineering, support, and customer success can all satisfy their local definitions of done while the end-to-end result disappears between them.

Continuity does not require one person to do everything. It requires a design in which the responsibility cannot vanish at a handoff. A mature FDE organization should steadily remove the need for heroics: platform teams eliminate repeated deployment failures, product teams create reusable connectors and controls, FDEs focus on discovery and the first production outcome, and enablement teams help customers take ownership.

The strongest FDE team is not the one with the most indispensable individuals. It is the one that repeatedly turns indispensable individual work into shared capability.

AI blurs the boundary again

Coding agents lower the cost of producing software. An FDE can move from customer conversation to end-to-end prototype much faster. A product engineer can respond to a specific customer without waiting for a separate field team. The distinction can no longer rest on who writes the code. It rests more on context and time horizon: who holds the customer's operational reality, and who is accountable for a general product that will remain coherent across customers?

Commercial models push in the same direction. Traditional SaaS is often priced per seat because the vendor supplies access while the customer remains responsible for the result. Model providers commonly charge for usage because tokens and calls are measurable, even if the business outcome is not. An AI agent may have enough autonomy to resolve a support request or complete a transaction, making outcome-based pricing more plausible.

Outcome pricing moves responsibility back to the supplier. What counts as “resolved”? Is a refund complete when the agent initiates it or when money reaches the customer? How is a sale attributed when price, advertising, and human staff also influenced it? These definitions become contract, system-design, audit, and operations questions. Someone has to turn a commercial promise into observable events and then explain failures.

Meurer's view of agent engineering therefore evolved. It remains a real technical discipline, concerned with tools, state, prompts, evaluation, guardrails, and runtime behavior. But when an agent engineer is accountable for what happens in the customer's operation, the work inherits the central logic of FDE. At the same time, product and infrastructure engineers are moving closer to customer evidence.

“FDE is dead; long live FDE” is not a contradiction. What dies is the hope that a single job description can contain every form of customer-facing technical work. What survives is the organizational function: stay close to reality, keep responsibility continuous, and carry what the field teaches back into the product.

Questions for reflection

1. At which handoff does your organization most often lose customer context?
2. If you removed the FDE title tomorrow, who would own discovery, production adoption, and product feedback?
3. Which recurring act of field heroism should already have become a platform capability?

Lesson 2 — Before You Hire an FDE Team

Kevin Bai, Anthropic · *Forward Deployed Engineering 101*

This lesson: a two-axis test for whether FDE fits a business; the platform requirement behind the model; what stays customer-specific and what must return to product; and how to avoid single-person dependencies.

The fastest way to misuse forward deployed engineering is to copy the title before understanding the economics.

Kevin Bai approached the subject from several generations of the function. At Anthropic he worked on applied AI. Before that he was the first person on Rippling's FDE team and helped grow it to roughly twenty-five people in a year; earlier still, he worked at Palantir. His talk was deliberately elementary in the best sense: begin with the conditions that made FDE rational, then ask whether those conditions exist in your own company.

His first piece of advice was not “hire great engineers.” It was: **ask whether you need the function at all.**

Why a platform is not the result

Palantir Foundry helps an organization bring data together, express it through an ontology, and build applications on top. That is technically valuable, but the description immediately invites a buyer's question: what does organized data do for my business?

An industrial executive does not buy cleaner tables as an end in themselves. A consumer-goods leader may care about shelf availability, sales throughput, or supply continuity. An energy operator cares about asset uptime and risk. Even if Foundry can support those outcomes, the customer has to translate a general platform into a particular operating system for its own business.

That translation carries a second cost. An application-building platform becomes more valuable as users learn to build with it. But training the customer's employees before they can produce the first result imposes a tax on adoption. The customer pays for the product, pays people to learn it, and waits for those people to become effective before value appears.

Palantir combined product and service to close the gap. The customer did not merely receive a software license or buy a block of consulting hours. It received a team that could learn the business, assemble a solution on Foundry, and own the path to an outcome. The platform provided leverage; the forward deployed team shortened the customer's path from capability to result.

The two-axis test

Bai described the decision with a two-by-two matrix. One axis is the technical complexity or flexibility of what is sold. The other is the technical ability of the buyer and user.

When a technical product is sold to technical users, the users can absorb complexity as part of their job. Developer platforms such as GitHub or observability systems such as Datadog may be sophisticated, but the users are engineers and infrastructure teams. The company may need documentation, developer relations,

solutions architects, or customer engineering. It does not automatically need an FDE to turn the product into an application.

When a relatively standardized product is sold to nontechnical users, configuration and conventional implementation may be enough. Collaboration and business applications can expose many options without inviting customers to build arbitrary software on top.

The distinctive FDE quadrant appears when a highly technical, highly configurable product is sold to a nontechnical buyer whose business outcome depends on customization. Foundry served industries that often lacked the concentration of software engineers found at a major technology company. An oil-and-gas operator has deep technical expertise, but its core pipelines are physical. It may not want to recruit and manage a platform engineering organization simply to turn a data platform into operational applications.

FDE bridges that mismatch: a powerful product on one side, a business buyer on the other, and substantial implementation judgment between them.

The framework is more useful than the title because it points to alternatives. If the buyer is technical, invest in a developer motion. If the product is standardized, invest in repeatable onboarding, sales-led implementation, and customer success. Choose FDE when the customer's ability to realize value depends on technical co-building that cannot yet be reduced to configuration.

A design partnership at enterprise scale

Early B2B startups often find product-market fit through design partnerships. The company works closely with a few customers, learns their problems, and builds a solution while the product is still uncertain. Bai's interpretation of Palantir was that it extended this intimacy into enterprise scale.

That sounds attractive until the maintenance bill arrives. If every FDE builds a separate application from scratch, the company accumulates repositories, one-off architectures, inconsistent controls, and customer-specific knowledge. Engineers spend their time preserving old promises. Revenue may grow, but gross margin and product coherence deteriorate.

Bai's blunt distinction was useful: without a shared platform, the company does not have an FDE motion. It has a development shop.

There is nothing inherently wrong with a services business. The problem is pretending that custom labor has software economics. The scalable form of FDE depends on **primitives**: reusable building blocks that can be composed into many customer solutions. A primitive might be authentication, a connector, a workflow node, an ontology object, a policy engine, an evaluation component, or an interface pattern. The FDE assembles and extends them rather than reinventing the base system.

Amazon Web Services offers an intuitive analogy. Customers could buy hardware, install operating systems, and build a database. Instead, AWS exposes compute, storage, databases, identity, networking, and many higher-level services as shared primitives. The components are granular enough to serve many architectures, yet substantial enough that users do not begin with a rack of servers.

The right granularity depends on the market. In a narrow vertical, the product may arrive sixty percent complete and allow the customer to customize the remaining forty percent. In a broad infrastructure market,

the primitives must be smaller because the provider cannot anticipate the final application. The economic test is the same: do repeated deployments consume shared capability, and does each deployment improve the set of shared capability available to the next?

Platform is a precondition, not an afterthought

Startups often reverse the sequence. A large customer requests custom work, so the company hires customer-facing engineers to capture the revenue. Management assumes the product can be generalized later. The first few deployments feel fast because talented people compensate for missing platform. Then every new customer increases the number of combinations that only those people understand.

Bai therefore offered two questions in order:

1. Must we sell something technically complex to a buyer who cannot or should not implement it alone?
2. Do we have—or are we prepared to build—a platform of shared primitives on which the field team can work?

A “yes” to the first and “no” to the second is not permission to ignore the platform. It is a warning that the business is about to finance product gaps with expensive human labor.

The platform does not have to be complete before the first FDE arrives. Field work can scout ahead. It discovers which connectors, data models, controls, and workflow components repeatedly block value. But the organization must have a mechanism and budget for turning those discoveries into maintained product. Otherwise “temporary” field code becomes the architecture.

What belongs to the customer, and what belongs to the product?

An audience question went to the heart of the operating model: which engineering changes stay on the deployed side, and which go back to the platform?

Bai's high-level rule was straightforward. Work that is genuinely unique to one customer's environment may remain there. Anything that can generalize should be generalized over time.

The difficult word is **genuinely**. Two requests rarely arrive in identical language. One customer says “billing account,” another says “legal entity,” and a third says “merchant hierarchy.” The surface differs, but the stable structure may be a reusable relationship among customer, contract, and payer. Generalization is not copying the first solution to everyone. It is finding the invariant underneath local vocabulary and expressing it as a primitive with explicit configuration.

A practical classification helps:

- **Environment-specific:** endpoints, credentials, network topology, local policy values, and mappings to an existing system.
- **Pattern-specific:** a recurring workflow shape that needs configurable steps, schemas, or controls.
- **Platform-level:** capabilities many patterns require, such as identity, audit, retries, evaluation, versioning, or rollback.
- **Disposable:** an experiment built only to answer a question, with no production promise.

Every deployed asset should have an owner and a planned destination. “We'll clean it up later” is not a destination.

AI expands the FDE quadrant

Why did the title become so popular in 2026? Bai argued that the software industry itself changed.

Agentic products are unusually malleable. Models can interpret language, call tools, generate code, and adapt behavior to context. A single platform can be configured into an insurance workflow, a legal workflow, a finance workflow, or an internal engineering system. That power makes the product harder to explain and makes implementation quality more dependent on the customer's data and process.

In other words, more software now enters the upper-right quadrant: technically complex, highly customizable, and sold to people who care about an operational result rather than the machinery underneath. Cheap code lowers the cost of building the last mile, but it also creates many more possible last miles.

This does not remove the platform requirement. It strengthens it. If an agent can generate a new integration in hours, an organization can accumulate fragile variants faster than before. AI increases the speed of divergence unless the team also improves primitives, evaluation, governance, and the route by which field learning returns to product.

Design against key-person risk

Customer intimacy produces knowledge that is difficult to document completely. One engineer knows why a permission is unusual, which executive cares about a metric, where a data feed lies, and what compromise allowed a launch. If that person goes on vacation or leaves the company, the apparent relationship between two organizations can turn out to be a relationship with one individual.

Bai encouraged collaboration among FDEs for exactly this reason. Important engagements should not have a single point of failure. Pairing, code review, shared architecture records, meeting notes, and planned rotations all test whether knowledge can move. The strongest test is operational: can another engineer safely make a change or handle an incident without calling the original owner?

This is not bureaucratic overhead. It is part of productization. Knowledge that cannot leave one person's head cannot become a repeatable deployment motion.

The minimum definition

Asked for the ideal profile, Bai reduced the role to a useful minimum: a customer-facing software engineer. The person should meet the company's engineering bar and be trusted in front of a customer.

That definition avoids two common mistakes. The first is hiring a persuasive relationship manager who cannot independently reason about systems. The second is hiring a strong engineer and assuming customer work means taking orders politely. An FDE must be able to challenge a requested solution, explain a tradeoff, narrow scope, and remain credible when the system fails.

The rest of the profile depends on the product and market. A data platform may need ontology and integration depth. An agent company may prioritize evaluation and workflow design. A regulated deployment may demand security and change-management experience. The stable core is technical judgment exercised in direct contact with the customer.

Before opening the requisition, however, return to Bai's two questions. Does the business truly occupy the FDE quadrant? And is there a platform strong enough to turn intimacy into leverage?

Questions for reflection

1. Which quadrant does your product occupy today, and is it moving as AI changes the product?
2. What percentage of a typical deployment is assembled from maintained primitives?
3. If the lead FDE disappeared for a month, what would the team be unable to explain or operate?

Lesson 3 — Customers Bring Solutions; FDEs Find the Problem

Vinoo Ganesh, Kepler · *How Forward Deployed Engineering Is Done at Kepler*

This lesson: why field engineering is a product strategy; how observation collapses oversized requirements; why company vocabulary belongs in the product model; and why every temporary patch should be treated as production code.

Vinoo Ganesh wanted to recover an older meaning of forward deployment. Before FDE became a go-to-market label, he argued, it was a way of discovering what product to build.

Ganesh spent seven years at Palantir building data infrastructure for defense and intelligence, including deployments in Iraq and Afghanistan. He created Project Frontline, a rotation that trained software engineers to work as FDEs; in his telling, roughly 350 people went through the program. He later built a related function at Citadel, where data and software products support investment decisions, and then co-founded Kepler, an AI platform designed to produce traceable, citation-grade financial analysis.

His thesis was sharper than "listen to customers." An FDE is an extension of the product function. The role is judged by its ability to identify an opportunity in the field, define the right problem, ship a useful solution, and turn what it learns into product leverage.

Phoenix: a correct system that failed on real data

The lesson began with a failure from Palantir's attempt to enter large-scale data storage around 2013. A project called Phoenix was designed in isolation. Under the assumptions used by its builders, the architecture worked.

The customer data did not share those assumptions.

Phoenix stored transaction information in time-bucketed Cassandra keyspaces. Apache Cassandra is a distributed NoSQL database; a keyspace is its top-level unit for organizing tables and replication. Time buckets made retention and roll-off seem manageable. Then the system encountered a blank date in a large financial dataset. The missing value defaulted to the Unix epoch, January 1, 1970. The retention logic attempted to create a bucket for every ten-minute interval between 1970 and 2013.

Ganesh said the result was about 2.3 million keyspaces. Given Cassandra's file-handle memory requirements in that design, starting the server would have required roughly fourteen terabytes of RAM. The product was dead on arrival.

The technical bug mattered less than the organizational bug. The team had designed a coherent system without co-building against the reality of customer data. A single blank field—a routine feature of long-lived enterprise systems—invalidated an elegant architecture.

The failure changed the product strategy. Field engineers were not merely there to install what the product team had designed. They were a mechanism through which product assumptions met evidence early enough to change the design.

A forty-seven-page requirement becomes one alert

Ganesh's first operating principle was: detect the real problem and ship the real thing.

A large dispatch and shipping company produced a forty-seven-page requirements document. It asked for a custom dashboard with fourteen metrics, drill-down alerts, and a substantial business-intelligence interface. The estimated development effort was about three months. The teams spent months discussing and scoping it.

Then someone went to the site and asked a dispatcher a basic question: what is the first thing you do with this information on Monday morning?

The answer was simple. The dispatcher checked whether trucks were late and, when necessary, called to redirect or replace a shipment. The valuable part of the proposed dashboard was not the dashboard. It was the moment when a late truck required action.

The team replaced the envisioned project with an alert that could be built in hours.

This was not a celebration of crude software. It was a lesson about the difference between a requested solution and an underlying problem. Customers describe artifacts they can imagine: a dashboard, a report, an agent, an integration. They may not state what happens after the artifact produces information, how the work is done today, or which single decision determines value.

Ganesh's three questions cut through that ambiguity:

1. What are you trying to accomplish?
2. What happens after you have the answer?
3. How do you solve the problem today?

The first clarifies the outcome. The second reveals the action loop. The third exposes the existing workflow, including the parts that should be preserved.

His bias for very small problems was deliberate. If a useful fix takes less than a day, build it, ship it, and close the loop. Do not inflate every observation into a strategic initiative. A quick solution can earn trust and, more importantly, access to the deeper work where product opportunities are visible.

The party that defines the problem gains unusual influence over the solution. FDEs become valuable in early sales not because engineers have suddenly become charming presenters, but because solving a real, narrow problem demonstrates judgment. The customer begins to share the constraints behind the formal request. Product discovery improves because the relationship moves from speculation to joint work.

Watch the workstation, not only the interview

Another Palantir customer received roughly a terabyte of data each day in Amazon S3, an object-storage service. The pipeline produced many CSV files. Moving the output to Apache Parquet, a compressed columnar format designed for analytics, would reduce storage and compute cost and shorten the pipeline.

A data-quality engineer opposed the migration for almost a year. Her explanation was that Parquet was worse and did not work for her. The argument sounded irrational from the pipeline team's perspective.

The reason appeared only when the team watched her work. She downloaded CSV files onto a Windows computer, opened them directly, and spot-checked the data. A CSV is a plain-text table that desktop tools can easily display. A Parquet file is efficient for machines but, at the time, had no viewer in her environment that she could simply double-click.

The team built a small Parquet viewer overnight. She approved the migration within days. Ganesh recalled the pipeline runtime falling from about seventeen hours to roughly two, alongside a major reduction in data cost.

The employee had not been resisting efficiency. She was protecting the only control that let her do her job.

Interviews rarely reveal that kind of fact. People describe the official workflow or rationalize a habit they no longer notice. Observation catches repeated actions: copying data between tools, switching tabs, pulling out a phone while a system waits, running the same check at the same time each morning, or reacting to a step with "I have to." Those are signals of friction and possible product opportunity.

Ganesh called physical access and a customer email address "data-mining permits." The phrasing is provocative, but the point is practical. The highest-density product information lives where work happens—in the conversation after a failed job, the spreadsheet opened beside the official application, and the moment a user abandons the designed path. A conference-room interview or survey can support field research. It cannot substitute for presence.

His rule was: residents get the truth.

Model the language of the company

Observation reveals not only actions but vocabulary. Inside one organization, sales may say "customer," operations "client," finance "billing entity," and developers "org ID." Each term may refer to a slightly different boundary. Integrations fail because a field that looks equivalent is not. Metrics disagree because teams calculate the same phrase differently.

This is why ontology appeared in Palantir's product design. Every organization can be described through nouns and verbs: the entities it recognizes and the operations people perform on them. The FDE's task is to locate overloaded terms, system boundaries, records that cannot be replaced, and translation points where one team's language becomes another team's schema.

The point is not to force all humans to speak like a database. Local language reflects legitimate differences in responsibility. Finance may need a billing entity that is not equivalent to the sales account. A useful ontology preserves those distinctions while making the relationships explicit. It gives applications and agents a controlled model for moving among them.

The issue is even more visible in AI. Teams use "agent," "skill," or "MCP" for systems with different boundaries. A function call wrapped in instructions may be called an agent; a complex autonomous workflow may receive the same name. If a deployment team does not define its nouns and verbs, engineers, users, and executives can agree verbally while imagining different systems.

Language becomes product infrastructure. When the product supplies a vocabulary that accurately represents the business, users begin to formulate later problems through that vocabulary. That is a powerful form of adoption—but it must be earned by modeling the customer's reality, not by branding every concept.

Every temporary fix wants to live forever

Ganesh's final story concerned a Groovy script he wrote as a quick fix for data retention. It was not designed as production software. It solved the immediate problem, spread through a customer with close to one hundred thousand employees, and remained in use long enough to become part of his identity. Friends even wore shirts with the script's filename at his wedding.

The humor masks a structural problem. A useful hack will be copied, scheduled, depended upon, and eventually treated as infrastructure. If it breaks, users do not care that its author described it as temporary. Mission-criticality is determined by use, not by the intention written in a ticket.

“Every hack goes into production” is an intentionally absolute rule that encourages the right caution. Before shipping, an FDE should ask:

- Will someone call me at 2 a.m. about this six months from now?
- What long-term tradeoff am I making for speed?
- Who can own the system when I leave?
- What happens when it fails?
- Should the capability enter the core product, remain a customer-owned asset, or be discarded after learning?

The lesson is not to stop shipping small fixes. Quick, concrete help is how field teams earn trust and discover reality. The lesson is to ship with an explicit lifecycle. A prototype must be marked and isolated as a prototype. A production bridge needs observability, ownership, and a retirement or productization path. If users will depend on it, build as if it will run for eighteen months—because it probably will.

Product leverage is the output

Ganesh contrasted two versions of the role. A weak FDE organization takes requirements, schedules research, records insights, and keeps the customer happy, yet never changes the product. A strong one redefines the problem, gets into the place where work happens, ships a narrow fix, owns its consequences, models the nouns and verbs behind the workflow, and converts the recurring structure into product leverage.

The measure is not how much the field engineer learned. It is what the company can now build, sell, or operate better because the engineer was there.

That perspective also clarifies a tension with Lesson 2. Bai described FDE as a go-to-market motion built on a platform. Ganesh insisted that an early-stage company should first treat FDE as product strategy. Both can be true at different stages. A mature platform can use FDE to carry product capability into the market. A young company must use the same contact to discover and harden the platform—or it will scale the wrong assumptions.

Questions for reflection

1. Which current requirement is actually a customer's proposed solution rather than the problem?
2. What repeated action would become visible only if someone watched the user work?
3. Which “temporary” customer asset has no owner or productization decision?

Lesson 4 — Map the Company Before You Automate It

Vasuman Moza and the Varick Agents team · *AI Tools for Forward Deployed Engineering*

This lesson: why execution is no longer the only bottleneck; how to model exceptions rather than the documented happy path; why AI must work with systems of record; and how an agent can augment the FDE by maintaining a machine-readable model of the company.

Vasuman Moza, founder and CEO of Varick Agents, framed the next constraint in enterprise AI as a question of depth: how far can a company understand and redesign a customer's operation without increasing human headcount at the same rate?

Models can now perform many end-to-end knowledge tasks. Tool protocols and agent harnesses let them browse, query APIs, manipulate documents, and execute multi-step work. Yet two organizations with the same department and the same enterprise software rarely operate in the same way. A finance team at a healthcare company does not behave like one at a SaaS company. Even two companies using NetSuite may have different approval paths, exception owners, account structures, and unofficial controls.

Execution is becoming abundant. A faithful model of the business is not.

The process in the manual is not the process in the company

Varick begins an engagement by narrowing a large enterprise to a department and embedding forward deployed engineers with the people who own its processes. In finance, that can mean accounts payable, accounts receivable, card reconciliation, banking, billing, and financial planning and analysis.

The engineers are not only documenting the standard path. Most company documentation describes what happens when the expected data arrives, the approver is available, and the systems agree. Production work is dominated by the cases where they do not.

The useful map sounds less like a procedure manual and more like this: Sarah handles the normal accounts-payable workflow, but when a purchase order and invoice do not reconcile she sends the case to Chris, who needs four days and a spreadsheet to resolve it. That exception path is often where cycle time, risk, and expert judgment are concentrated. It also differs from one company to the next.

An FDE therefore needs two workflow diagrams.

The first is the **declared process**: the approved stages, formal systems, policy, and intended owner. The second is the **observed process**: messages, spreadsheets, side conversations, retries, escalations, and local decisions that actually move the work. The difference between the two is not merely inefficiency. It may encode controls the formal design failed to express.

If an AI system is trained only on the golden path—the route in which all assumptions hold—it will look excellent in a demonstration and fail exactly where the organization needs the most help.

Do not attach AI to a broken process

Moza argued that companies often deploy AI as an extra step on top of an existing workflow. A model summarizes a document, drafts a message, or recommends an action, but the surrounding process stays unchanged. The user still transfers the output manually, resolves the same exception, and waits for the same approval. The visible AI step becomes faster while the end-to-end cycle barely moves.

Process re-engineering asks a different question: if we designed the workflow around the capabilities and limits of AI, how would responsibility be divided?

The answer should not be “the agent does everything.” A useful redesign might make four steps autonomous, place three behind human review, and leave one entirely human because the risk is too high or the required judgment is not sufficiently repeatable. The division depends on error cost, reversibility, available evidence, and the organization's willingness to change.

Adoption matters as much as theoretical efficiency. If employees have spent years using an eleven-step process, collapsing it overnight into an opaque button can destroy their ability to understand or control the work. A redesigned workflow should remove unnecessary coordination while preserving visible control points. Users need to know what the agent did, why the item reached them, what evidence supports the recommendation, and how to correct or escalate it.

The FDE's job is therefore not to automate the diagram. It is to redesign the operating agreement among people, agents, and systems.

Systems of record will not disappear for your agent

Enterprise operations are anchored in systems of record: the applications that hold authoritative financial, customer, inventory, or audit data. ERP systems such as SAP S/4HANA, Oracle NetSuite, and Microsoft Dynamics; CRM platforms such as Salesforce; and company-specific databases represent years of migration, policy, and organizational learning.

Moza quoted a customer that had spent five years and five million dollars moving to NetSuite. Telling that company to abandon the system in order to adopt a new AI tool is not a deployment plan.

The realistic architecture places agents on top of or alongside existing records. The agent may gather context from several systems, make a recommendation, and write an approved action back. The system of record remains authoritative. This preserves downstream processes, access controls, and audit expectations while allowing the workflow above them to change.

It also makes integration a product concern rather than plumbing to be delegated. The agent needs to understand object identity, permission boundaries, validation rules, and the difference between reading an ERP and committing a financial event. “Connected to NetSuite” says very little unless the deployment specifies which records, actions, roles, and failure paths are supported.

Varick described its own operating layer as a platform for creating and monitoring agents with governance and evaluation built in, while leaving core systems in place. Whatever vendor is used, the design principle is broader: **AI should not create a parallel universe in which the official state and the agent's state drift apart.**

Build an agent for the FDE

Deep deployment creates a scaling problem. A good FDE may receive constant emails, hundreds of pages of documentation, meeting notes, slide decks, and competing requests from process owners. Accounts payable depends on accounts receivable; reconciliation depends on banking; planning depends on all of them. The engineer has to preserve context across the network without treating the most persistent stakeholder as the highest priority.

Varick's response was an internal “FDE agent”—a system intended to augment the scarce engineers who can combine technical depth with customer communication.

An engineering leader identified in the talk as JD described three stages.

1. Engagement assistant. The system ingests meeting notes, documents, presentations, email, and other engagement context. It can answer practical questions: who owns this process? Are two differently spelled names the same person? Where was this exception described? The goal is not a generic summary. It is reliable retrieval and synthesis for the people running the deployment.

2. Workflow copilot. The assistant lives inside the platform while the FDE constructs a workflow. It can point out a missing edge case, identify an owner, or warn that the proposed route does not match what was learned during interviews.

3. Autonomous maintenance assistant. The long-term idea is to handle small requested changes. If a customer asks to send a quality-control report to a different address, the system could interpret the request, query its model of the company, propose or make a bounded workflow change, test it, and route it for approval. Human FDE time remains available for discovery and high-value design.

This sequence matters. The system should earn autonomy by first becoming good at context, then good at advising within a controlled surface, and only later good at changing workflows.

Give the company a machine-readable operating model

An engagement assistant needs more than a folder of documents. Varick represents company operations as a dependency graph. The specific storage technology is less important than the abstraction: tasks, people, systems, and approvals are nodes and relationships; downstream work depends on upstream state; cycles reveal rework, escalation, or an ambiguous process.

The graph acts as a machine-readable model of how the company operates. It lets the system ask whether two names refer to the same person, identify duplicate steps, trace why an approval is required, or find a violation in an intended dependency chain.

There are two distinct model problems.

First, given the right evidence, can the system produce a concise and accurate process description? General-purpose frontier models can be verbose and fail to distinguish the detail a decision-maker needs from the detail that can be safely omitted. Varick described post-training models on examples of normalized process analysis to improve that judgment.

Second, can the system retrieve the right evidence from a large, messy graph? This is not solved merely by giving the model a longer context window. The platform used specialized tools for graph traversal and a reinforcement-learning environment to train better selection: resolve person identity, follow dependencies, detect cycles, and gather the context relevant to a particular process question.

This split is useful for any enterprise agent. Poor output may come from weak reasoning over good context, or from confident reasoning over the wrong context. Evaluation must test both retrieval and analysis.

Automate the FDE, or amplify the FDE?

Moza used ambitious language about automating knowledge work, but the system he described points toward amplification rather than immediate replacement.

The agent handles context maintenance, repeated analysis, and small bounded changes. The human remains responsible for interviewing process owners, noticing contradictions, deciding what should change, negotiating adoption, and understanding risk. Those are not residual tasks left over because the model is temporarily weak. They are the work of creating the operating model against which an agent can be useful.

Over time, more maintenance can move to the system. But every increase in autonomy requires a stronger representation of the company, better permissions, and more reliable evaluation. Otherwise the FDE agent becomes another participant with a partial, outdated picture of the workflow.

The strategic question is not “can AI execute this step?” It is “does the organization have a sufficiently accurate and governed model of the work for AI to execute the step without creating invisible damage?”

Questions for reflection

1. Where does the observed process diverge most sharply from the documented process?
2. Which system must remain the source of truth after an agent is deployed?
3. What would an internal FDE assistant need to know before it could safely propose a workflow change?

Lesson 5 — The Scarce Engineering Skill Is Saying What Not to Build

Leo Mehr, Ramp · *How Forward Deployed Engineering Is Done at Ramp*

This lesson: why scoping is an engineering discipline; how AI can scale the FDE pipeline; why context, rubrics, and human taste remain necessary; and the two symmetrical ways an agent-native field team can fail.

Leo Mehr reduced Ramp's FDE experience to two rules: **always be scoping** and **scale with tokens**.

He joined Ramp when the FDE team consisted of two engineers. About two and a half years later, he led an organization of roughly thirty across forward deployment, developer API, and newer AI services. Ramp's FDEs sit inside engineering and help its financial operations product work for large enterprise customers. Their job is not to say yes to every request. It is to help the customer succeed by identifying the right thing to build—and then to increase how much of that work can be performed by models and agents.

The rules must be held together. Scoping without agent leverage becomes too slow. Agent leverage without scoping produces failure at machine speed.

The Friday-night SAP request

Mehr described a familiar enterprise escalation. Late on Friday, a sales representative reports that a strategic customer will not sign unless Ramp builds an integration with SAP S/4HANA, SAP's enterprise resource planning suite.

The default engineering response is to locate the API documentation and begin estimating work. A trained FDE pauses before accepting the request's premise.

What is driving the deadline? Is the customer actually blocked, or is a sales quarter ending? Who will use the integration? What exact data or action is required? Has the team exhausted available workarounds? Can a temporary manual process bridge the gap? Does the customer have technical resources that can use an existing API? Which other customers or prospects need the same capability?

These questions are not resistance. They separate urgency from importance and a named solution from the smallest outcome that unlocks value. They also expand the evidence beyond a single deal. An integration that several customers need may belong on the product roadmap. A request tied to a local implementation can remain customer-specific. A sales deadline should affect sequencing only after the team understands what it is sequencing.

The phrase “always be scoping” matters because scope is not settled once at kickoff. New facts appear during implementation. A stakeholder changes. An assumed API behaves differently. A workaround turns out to be acceptable. The FDE continuously asks whether the current plan is still the shortest responsible route to the result.

The missing qualifier

Ramp learned the cost of untested assumptions through a mobile reimbursement feature. The core mobile team was busy, so two FDEs learned enough iOS and Android development to build the feature on both platforms. After weeks of work they asked the customer for Android beta users.

The customer required every employee to use iOS.

The requirement had contained an invisible qualifier: “on mobile” meant “on the only mobile platform this organization permits.” The engineers solved a broader technical problem than the customer had.

This is an unusually clean example of wasted effort, but the same error appears in less visible forms. “All employees” may mean one region. “Real time” may mean before the next morning. “Integrated” may mean a daily file, not bidirectional API access. “Automated” may still require approval above a threshold.

Good scoping hunts for the missing qualifier. Before building, write down the user, environment, channel, frequency, permissions, non-goals, accepted workaround, and success event. Then ask which statement would make half the proposed work unnecessary if it turned out to be true.

Scale with tokens, not only headcount

Scoping is valuable, but human-only scoping does not keep pace with the expanding volume of customer requests. Mehr's second rule is to redesign the FDE's own workflow around models.

Consider the complete pipeline: gather customer context, clarify the request, inspect the existing product, define scope, write a specification, implement the change, evaluate it, and ship. Each stage contains knowledge work that an agent can assist or, under the right controls, perform.

Ramp began with request intake. Account managers, customer-success managers, and salespeople post enterprise blockers in an internal channel linked to a Notion workflow. Input quality varies from a complete description to a single line such as “we need an SAP integration.” Previously, FDEs read each request, searched product context, and conducted multiple rounds of clarification.

An initial Notion agent asked a few questions as soon as a request arrived. Latency fell from hours or days to seconds. Submitters engaged while the need was still fresh. A later version conducted several rounds of follow-up until it had enough information to propose a specification. Mehr estimated that the system saved around twenty percent of the team's scoping time, framing the number as an internal judgment rather than a controlled measurement.

The deeper value was not the percentage. The agent made good intake behavior immediate and consistent. It could insist on basic evidence even when human FDEs were busy, creating better material for later judgment.

An FDE pipeline is an agent system

Request clarification is only the first stage. A mature version could use specialized agents for product research, dependency analysis, specification drafting, implementation, and validation. The final construction of a medium-sized feature may become easier as coding models improve. Mehr expected the difficult middle to

remain: converting incomplete customer evidence into a correct product decision and a specification that captures all necessary context.

Treating this as a pipeline creates several engineering requirements.

Context. The agent needs product architecture, historical decisions, customer conversations, support history, documentation, and the tacit knowledge a product manager carries. A larger prompt does not automatically produce the right context. The system needs retrieval, memory, tools, and explicit rules about which sources are authoritative.

Evaluation. Each stage needs a definition of quality. A request is not "good" because it is long. A specification can be checked for user, outcome, non-goals, permissions, dependencies, failure behavior, and success metrics. Implementation can be checked against tests and operational controls. A rubric turns taste into inspectable criteria without pretending judgment is fully mechanical.

Traceability. When an agent changes a scope or produces a specification, the team should be able to see the evidence it used, the questions it asked, and unresolved uncertainty. Otherwise automation hides the reasoning that field work exists to improve.

Human feedback. The system should record which drafts were accepted, corrected, rejected, or later associated with a production failure. Those decisions become the training and evaluation material for the next version.

Taste remains an ownership function

Mehr did not describe a future in which the FDE disappears after the pipeline is automated. The human retains responsibility for taste and judgment over the final output.

Taste here is not a mystical aesthetic. It is the ability to recognize whether a technically valid answer fits the product, the customer, and the long-term system. A specification can include every required section and still commit the company to the wrong workflow. A generated feature can pass tests and still create an incoherent interface. A correct integration can encourage the customer to depend on data that is not authoritative.

Rubrics make recurring standards explicit. They do not decide which tradeoff the company should accept. The FDE remains accountable for that decision and for improving the system when the decision recurs.

Two failure modes

The first failure is a **token-maxxing slop cannon**: an organization uses agents to generate specifications and code at enormous volume without controlling scope. It automates the production of the wrong work. Output rises while customer outcomes, maintainability, and product coherence fall.

The second failure is the artisan bottleneck: a team scopes every request beautifully but refuses to automate its own repeated work. Quality may remain high for a while, but response time and cost cannot compete with agent-native organizations. The best engineers spend their days performing intake and synthesis that could have been delegated.

The future of FDE requires both disciplines. Scope determines what deserves execution. Agents increase the capacity for execution and analysis. Evaluation connects the two.

A useful operating loop is:

1. Let the agent gather and challenge basic context immediately.
2. Let the human narrow the goal and decide the important tradeoffs.
3. Let specialized agents assemble evidence, draft, implement, and test.
4. Let the human accept responsibility for the product decision.
5. Feed corrections and production outcomes back into the system.

“Always be scoping” is what prevents speed from becoming waste. “Scale with tokens” is what prevents judgment from becoming a scarce manual service.

Questions for reflection

1. What unverified qualifier could eliminate half the work in your most urgent request?
2. Which stage of your field workflow is repeated enough to automate first?
3. What evidence would tell you that the automated pipeline is producing more output but worse decisions?

Lesson 6 — When Code Is Cheap, Restraint Becomes Expensive

Sunny Rekhi, Decagon · *How Forward Deployed Engineering Is Done at Decagon*

This lesson: how an FDE model changes during hypergrowth; why configuration and product engineering separate into different lanes; how domain knowledge compounds; and why every repeated customization should move toward self-service.

Decagon builds AI agents for customer service across voice, email, messaging, and other channels. The agents do more than answer questions. They can take actions through customer systems, follow brand policy, hand cases to human representatives, and eventually support revenue-generating interactions.

That product makes forward deployment unavoidable. The same underlying agent has to represent different brands, policies, customer intentions, risk tolerances, backend systems, and definitions of success. A password reset and a lease renewal are both “customer interactions,” but their tools, evidence, failure costs, and commercial value are different.

Sunny Rekhi, Decagon's CTO of Forward Deployed Engineering at the time of the talk, described how the operating model changed as the company grew from roughly fifty to five hundred people in a year. Hypergrowth made a central tension impossible to ignore: the team had to move quickly for large customers without turning every deployment into a black box that only its original builder understood.

Two kinds of forward deployment

Rekhi separated the work into two broad lanes.

The first configures the agent for the enterprise. The team defines how it should speak, what user intentions it should handle, what must be transferred to a human, and which actions it can take. It connects backend systems and shapes policy. This resembles training a new employee, except the operating instructions also have to be formal enough for evaluation and version control.

The second lane handles product gaps discovered at the front line. A large enterprise asks for a capability that the current platform does not provide. The deployed engineer has to determine whether another customer will need the same thing soon, then build or influence a solution that serves the larger product.

At an early stage, one “agent software engineer” could do both. The same person sat with the customer, configured behavior, built integrations, and changed the platform. At five hundred people, the role divided into specialists: agent builders with deep intuition for configuring and testing the agent, and agent software engineers who bring enterprise needs into the core product.

The reporting line and engineering bar remained close to product engineering. That choice expresses a belief: a request from a major customer is often product evidence, not a ticket for a separate implementation department.

The temptation of instant custom code

AI coding tools make the wrong move unusually tempting. A customer asks for a feature. The account matters. An engineer can prompt a coding agent and produce a one-off implementation quickly. Everyone experiences relief.

The cost appears later. Another customer needs a slightly different version. Prompts and patches accumulate. The agent's behavior can no longer be explained through the product's supported controls. The customer cannot safely own or modify what was built. The system becomes brittle precisely because the original implementation was so easy.

Rekhi argued that restraint is now a scarce engineering skill. The question is not only whether the team can satisfy the request. It is whether the solution remains understandable, supportable, and useful to customers B, C, and D before they make the same request.

This does not mean waiting for a perfect abstraction. It means recognizing the long-term shape of the decision. A one-off may be justified to prove value, but it should be built within a known extension surface, instrumented, and classified for productization. If the only way to change the agent is to ask its original engineer to edit a private tangle of prompts, the deployment has failed the ownership test.

Decagon's stated design goal is that customers can understand and own their agents. That turns clarity into an architectural requirement. Policy should live in a visible configuration model. Actions should have explicit schemas and permissions. Evaluation should reveal how a change affects behavior. The system should not depend on folklore about which patch counteracts which prompt.

Write success down before building

Speed does not begin with coding. It begins with a shared definition of success.

Rekhi advised teams to establish that definition in the first conversations and, ideally, put it in writing. For a customer-service deployment, it may specify the channel, the customer intentions in scope, the target containment or resolution rate, the quality threshold, the escalation conditions, and the business metric the project should move.

The written definition protects both sides from silent drift. A buyer may imagine a voice agent handling every call while the project team believes it is launching email for three intentions. A vendor may optimize automation rate while the customer cares about customer satisfaction or revenue. Without an explicit target, every demonstration can look promising and every production result can be disputed.

Written success also disciplines the desire to start immediately. Code generation reduces the visible cost of an early mistake, but it does not reduce the cost of integration, testing, stakeholder alignment, and change management. Building the wrong agent faster still delays the first result.

A strong success statement contains at least:

- the user and channel;
- the intentions or workflow included;
- actions the agent may take;

- handoff and human-control rules;
- the business and quality metrics;
- the observation period;
- explicit non-goals.

The statement should be narrow enough that both parties can recognize completion.

Domain knowledge compounds

As Decagon deployed across industries and company sizes, it began staffing people with repeated vertical experience. A team that has worked with several financial institutions knows more than the vocabulary. It recognizes recurring compliance concerns, identity structures, approval patterns, and plausible success metrics. The next financial customer can begin from a better set of questions.

This is not an argument for rigid industry templates. Two banks may still differ materially. It is an argument that field learning should compound at the level of teams and systems. The company should preserve playbooks, examples, evaluation cases, ontology patterns, integration lessons, and decision records so that experience survives personnel and customer boundaries.

Domain fluency also increases credibility. An FDE who understands the customer's nouns and constraints can challenge a requested solution without sounding dismissive. The conversation moves sooner from “can the model do this?” to “which part of this workflow produces the highest return under your controls?”

Prove value before expanding the boundary

Large enterprises often arrive with the whole kitchen sink. They can imagine the agent supporting every channel and handling a long catalogue of intents. A broad vision may be strategically correct and operationally disastrous as a first deployment.

Rekhi recommended finding the shortest route to provable value. Choose a useful slice, put it into production, demonstrate the result, then expand. In customer service, that may mean one channel and a narrow set of high-volume intentions. It should be meaningful enough to change work, but constrained enough that integrations, evaluation, and escalation can be completed in weeks rather than indefinitely.

The sequence creates trust. Once the agent reliably resolves a real support flow, the team can add channels, actions, or proactive revenue use cases. Decagon's example with Hertz was that the relationship began with complex inbound support and later expanded into proactive customer communication around lease renewal or extension. That description is a speaker-provided customer example, but the product logic is general: land with a bounded problem, learn the systems and relationship, then expand into adjacent value.

This is different from a shallow proof of concept. The first slice should enter production with real users, monitoring, and ownership. Its boundary is narrow; its standard is not.

From custom to self-service

Rekhi repeated one phrase as a design rule: **custom becomes self-serve**.

Early in Decagon's history, engineers built customer integrations one by one. After enough repetition, the team recognized that the recurring work needed a product surface. What had required custom code could be configured by an agent builder or eventually by the customer.

Self-service is not simply a settings page. It requires the product to expose a stable concept, validate inputs, guide users toward safe choices, test the resulting behavior, and make failures understandable. The hidden engineering remains substantial. The difference is that it is paid once in the platform rather than again in every deployment.

A practical productization funnel is:

1. A field team handles the first instance and records the full context.
2. A second instance reveals which parts are stable and which are local.
3. Product and engineering design a reusable primitive or configuration model.
4. Agent builders use it without core engineering changes.
5. Customers can use it with guardrails, evaluation, and documentation.

Not every task should reach the last stage. Security-sensitive integrations may continue to require engineering. The goal is to move the boundary deliberately rather than allowing repeated manual work to become permanent.

Act as an advisor, not an order taker

Because a forward deployed team sees similar operations across many companies, it can know something the customer cannot easily know: which use cases tend to produce value and which deployment choices later create trouble.

Decagon analyzes historical support data and can recommend which intents to automate first. That recommendation may differ from the request that opened the conversation. The FDE still has to execute, but the role includes advising the customer about sequence and return.

Advice must be grounded in the customer's evidence. Cross-customer pattern recognition is valuable, but it can become arrogance if the team assumes all companies are alike. The responsible form combines prior knowledge with local data: “In similar deployments this category produced the fastest return; your volume and escalation data suggest the same—or show why it does not.”

At scale, this advisory judgment, the platform funnel, and vertical memory work together. The team moves quickly because it is not relearning the field from zero. It remains a software company because repeated work flows upstream. It keeps customer trust because the first production result is explicit and measurable.

Questions for reflection

1. Which current agent behavior can only be changed safely by its original engineer?
2. What exact result could your deployment prove in the smallest production slice?
3. Which manual step has repeated enough times to justify a self-service product surface?

Lesson 7 — The Customer Environment Is the Highest-Fidelity Evaluation Set

Jia Wu, Cognition · *How Forward Deployed Engineering Is Done at Cognition*

This lesson: why writing code is only a fraction of enterprise software work; how deployment maximizes the overlap between product and problem; why the field is an evaluation system; and why ROI must be measured in accepted work and delivery outcomes rather than tokens.

When Cognition introduced Devin in 2024, the company presented an autonomous software-engineering agent and reported performance around thirteen percent on SWE-bench, a benchmark built from real GitHub issues and their corresponding fixes. The announcement made the future feel close and, at the same time, exposed how far a benchmark result was from reliable work across an enterprise.

By 2026, Devin appeared through several surfaces, including command-line and development interfaces as well as a cloud agent able to work asynchronously. Jia Wu, a deployed engineering lead at Cognition, argued that improvement in the model alone did not explain enterprise use. The deployed engineering motion made the product real by mapping it to high-value software workflows, automating the operation around it, and carrying what failed back into the roadmap.

Maximize the overlap

Wu described product-market fit as the overlap between two circles: the capabilities a company has built and the problems customers need to solve. Deployed engineering tries to enlarge that overlap from both sides.

On the customer side, FDEs study the actual software-development lifecycle. They examine backlogs, delayed migrations, missing tests, incident triage, review queues, maintenance work, and code that repeatedly fails to ship. The goal is not to distribute an agent and wait for employees to invent use cases. It is to find initiatives where the agent's capabilities can change a meaningful constraint.

On the product side, the field team returns evidence. Similar challenges appear across enterprises in different forms. Workarounds reveal a missing feature. Repeated failures reveal a weak evaluation. A capability that works only after an FDE supplies hidden context reveals a product gap. The deployed engineer helps the product team decide which shapes are common and which are unique.

Solving the customer problem is only half the loop. If the product remains unchanged, the next deployment begins with the same friction.

Coding is perhaps twenty percent of the problem

Wu made an intentionally rough estimate: writing code may account for only about twenty percent of the enterprise problem.

The exact percentage is not important and should not be treated as a general measurement. The decomposition is. Software has to be specified, tested, reviewed, secured, deployed, operated, and maintained. Legacy repositories have inconsistent conventions and dependencies. Work must pass organizational controls.

A patch becomes valuable only when it is accepted into the codebase and changes the intended system without unacceptable risk.

Modern models are increasingly competent at generating a code block when given sufficient context. The harder enterprise question is how to construct that context and the surrounding lifecycle. What event should trigger the agent? Which repository and branch may it use? How are credentials provided? What tests define success? Who reviews the change? What happens when a dependency is unavailable? How is the agent prevented from converting a local fix into a systemic defect?

Deploying an agent without answers to these questions is what Wu called token-maxxing: consuming inference and producing activity without a credible path to outcome.

Automate yourself out of manual operation

Cognition's FDEs may divide a day between customer conversations and hands-on engineering. The conversations identify the highest-leverage initiative. The engineering should then do more than run the agent manually for a few demonstrations.

A completed deployment makes the workflow event-driven and repeatable. A backlog item, alert, dependency update, or other signal can trigger the appropriate automation. The agent receives the environment, tools, and instructions it needs. Humans review at defined control points rather than relying on the FDE to start and supervise every session.

This is what it means for the FDE to automate themselves out of the job. It does not mean the customer no longer needs engineering judgment. It means the deployed system no longer depends on the field engineer as a human scheduler and context courier.

The transfer test is practical. Can the customer's team operate, observe, and improve the workflow after the FDE leaves? If every successful run still begins with a message to the vendor's engineer, the organization has purchased a managed demonstration.

Production creates the best evaluation data

Benchmarks are useful because they make comparison possible. SWE-bench, for example, asks whether a model can produce patches for real repository issues that pass project tests. But a benchmark cannot reproduce one company's codebase, private tools, security controls, architectural history, and definition of acceptable work.

The customer environment is therefore the highest-fidelity evaluation set available to an enterprise agent company. It contains the real distribution of problems and the real consequences of failure.

Field evidence can be organized at several levels:

- **Task validity:** did the generated change satisfy the stated issue and pass deterministic checks?
- **Organizational acceptance:** did reviewers approve and merge it, or did the change create unacceptable maintenance burden?
- **Workflow impact:** did the agent reduce queue time, clear a migration backlog, or increase the number of valuable changes shipped?

- **Business effect:** did the software initiative improve cost, risk, revenue, or time to market?
- **Generalization:** does the same failure shape appear across customers, and should it alter the product or evaluation suite?

The FDE turns these outcomes into product information. A rejected pull request is not merely a customer incident; it may be an evaluation case. A repeated inability to locate repository policy may justify a new context mechanism. A successful workflow across several companies may become a supported automation.

This loop de-risks the roadmap. Product priorities are no longer based only on feature requests or model benchmarks. They are tied to evidence that a capability changes production work.

Stop proving value with tokens

Early agent deployments often measured usage: sessions, tokens, generated lines, or hours of nominal agent activity. These are useful operational metrics. They do not prove value.

Wu presented several anonymized and public examples from Cognition to illustrate a move toward delivery measures. The talk described one three-month engagement as producing agent capacity comparable to roughly 150 additional contributors; it also cited an approximately eighty-two-percent reduction in a delivery timeline and close to twice as many pull requests compared with earlier tool use. Other examples involved data migrations and legacy languages such as COBOL and JCL. These figures are company-reported examples, not controlled evidence that can be generalized to another environment.

Their purpose was to show a measurement ladder.

At the bottom is **activity**: sessions ran, tokens spent, tasks attempted. Above it is **valid output**: tests passed, pull requests opened, reviews completed. Next is **accepted output**: changes merged and used. Above that is **flow**: lead time, backlog age, incident time, and release frequency. At the top is the business outcome attached to the engineering initiative.

Each higher level is harder to attribute, but closer to what the customer purchased. A credible deployment reports both: activity explains how the system was used; outcomes explain why the use mattered.

From individual acceleration to organizational throughput

An IDE assistant can make a developer faster at the moment of coding. That is valuable, but it does not automatically accelerate the organization. Review may become the new bottleneck. Generated changes may increase coordination and test load. Nontechnical stakeholders may still be unable to turn an initiative into validated work.

Wu's stronger claim was that enterprise value comes from orchestrating the system, not simply accelerating one user. A deployed agent should participate across the lifecycle and fit the organization's technical and nontechnical roles. The output must arrive in the forms the organization can accept: a reviewed pull request, a completed migration slice, a triaged alert, or an updated test suite.

That is why deployment requires more than installing a CLI or IDE extension. The work includes event design, permissions, shared context, evaluation, and change management across the engineering system.

Who succeeds in the role?

Cognition looked for breadth with a genuine spike. FDEs need enough business, process, customer, and technical understanding to move across the engagement. They also need a depth that makes them the expert in some critical part of the room.

The spike can vary. A product manager with strong system sense may be good at fitting capabilities into a coherent workflow. A founder may be comfortable moving from ambiguous customer need to working system. A software engineer may bring deep technical authority. Business fluency can be learned; so can customer communication. But the role cannot be credible if nobody on the deployed team can reason deeply about the technology being changed.

The differentiator is relentless curiosity about why. Why does the problem matter to the business? Why is this the right initiative? Why did the agent fail here? Why should the evidence change the roadmap for everyone else?

Curiosity keeps the FDE from confusing adoption with usage and output with value. Customer commitment keeps the questions attached to real consequences.

Questions for reflection

1. What percentage of your agent workflow happens after code is generated?
2. Which customer failure should become a permanent evaluation case?
3. What metric above raw activity can your next deployment credibly move?

Lesson 8 — From Coding Agent to Software Factory

Eno Reyes, Factory · *How Forward Deployed Engineering Is Done at Factory*

This lesson: why a model is only one component of an autonomous engineering system; how verification determines agent readiness; why long-running work needs bounded task containers; and how a deployed team can demonstrate the future without building a theme park.

Factory did not want its deployed engineers to become a modernization consultancy. If a customer had received a large quote to migrate a codebase, Eno Reyes explained, the company did not want to take over the project merely because Factory's product might be used to perform it. Services revenue can be substantial, but doing the migration for the customer does not necessarily improve a scalable product.

Instead, Factory described its deployed engineers as the **tip of the spear**. They carry information from engineering leaders and working developers inside major customers back into the product. They help the organization build a new operating model for software development, then make the platform increasingly capable of assembling itself in that environment.

The object being deployed is not just a coding agent. It is a software factory.

The software factory already exists—poorly instrumented

Every engineering organization has an implicit production loop.

Signals arrive as customer conversations, bug reports, internal messages, incidents, regulatory requirements, or executive decisions. People triage and prioritize them. Plans become changes to a codebase. Changes pass through review, quality assurance, security analysis, static analysis, linting, type checking, and tests. Approved work is deployed and monitored. The running software produces new signals, and the loop begins again.

Most companies already perform every stage. The problem is that state and evidence are fragmented across issue trackers, chat, repositories, CI systems, security tools, and people's heads. Handoffs depend on manual coordination. Feedback from production does not reliably update planning or agent instructions.

A **software factory** makes the loop explicit and observable. Agents can participate in triage, planning, implementation, code review, testing, security analysis, documentation, deployment, and incident response. The goal is not to remove humans from engineering the system. It is to let a unit of intent move through a designed pipeline while humans govern objectives, constraints, and exceptions.

Factory's 2026 product description emphasizes several properties. The agent harness is model-independent so the organization can choose models according to cost, speed, and performance. The enterprise retains traces and context. Controls govern where data and actions can flow. Deployment can range from managed cloud to isolated or air-gapped environments. Every stage can feed learning back into the same system.

These properties matter because a factory is infrastructure, not a chat session. A company that cannot inspect the traces, change the model, control the data, or preserve the learning generated by its own workflows does not fully own the production system.

The model is not the system

A capable model can still fail in a codebase that provides weak feedback. It may produce code that looks plausible but violates a hidden convention, misses a security rule, or cannot be built in the local environment. More intelligence helps; it does not create the missing signals.

Reyes defined **agent readiness** largely through deterministic validation loops. A compiler, unit test, type checker, linter, security scan, or policy rule gives a clear and repeatable result. The more dense, relevant feedback a repository provides, the longer an agent can work on a complex task without human interruption.

This changes how organizations should prepare for autonomy. The first investment may not be a better prompt or model. It may be repairing the test suite, making the development environment reproducible, documenting repository policy, exposing safe tools, or converting a subjective manual check into a reliable validator.

Verification is not limited to code correctness. An enterprise workflow may also need evidence that a migration preserved data, a generated interface has no visual regression, a dependency license is allowed, or a deployment stayed within cost and latency budgets. What cannot be validated becomes the boundary at which humans must inspect or the system must stop.

Reyes reported that automated tools might fix thirty to forty percent of obvious readiness issues, while the remainder requires workflow and organizational change. The percentages are a speaker estimate, but the distinction is important. A linter configuration can be added mechanically. A team may still resist extremely strict checks because they interrupt established development flow. Readiness is partly a technical property and partly an agreement about how work will be produced.

Missions: bounded work with a verifiable end

Factory calls its long-running, multi-agent task container **Missions**. The essential design is broader than the product name: define a bounded task, define what completion means, and let the system continue working until the verification conditions are satisfied or it reaches a stop condition.

This differs from an open-ended request such as “modernize the codebase.” A bounded migration might specify the repositories, target framework, compatibility requirements, required tests, performance threshold, and artifacts to produce. The agent can decompose the task, work in parallel, test changes, revise failures, and keep a trace. Human involvement is concentrated in planning, governance, and unresolved exceptions.

Long horizon is useful only when the end state is inspectable. Without verification, more inference means the agent can travel farther in the wrong direction. With dense feedback, each failed attempt supplies information that keeps the work near the objective.

Reyes described very large autonomous migrations and work in scientific and financial domains. These examples are claims from the talk and should not be read as a guarantee that any large task can be solved by adding validators. The more defensible principle is narrower:

> Autonomy grows with the system's ability to observe and verify the consequences of action.

Where completion cannot be defined, the task must be reframed, new validators must be built, or human judgment must remain in the loop.

The hard work begins after the low-hanging fruit

Agent readiness scans reveal obvious improvements: missing tests, inconsistent types, static-analysis failures, undocumented setup, and reproducibility problems. Coding agents can fix many of these. The remaining work touches the way teams operate.

Suppose an agent-generated change passes automated tests but creates visual flicker in a terminal application. If no validator can detect the behavior, autonomy stops at human inspection. Building a reliable visual test becomes an engineering project. The person is not doing the agent's coding task; the person is improving the factory that will validate future coding tasks.

This is the change in abstraction Reyes emphasized. Engineers move from directly manipulating every software artifact toward designing and maintaining systems that produce software. They create validation loops, tools, context pipelines, model-routing policy, governance, and observability. Human responsibility expands upward rather than disappearing.

Developer-experience engineers may be especially prepared because they already build environments and “paved roads” that enable other engineers. Product managers who become technically deep can help formalize intent. Systems thinkers who enjoy modeling flows and closing loops can bridge the organization and platform.

Connect engineering output to a business result

A software factory should not be justified by the number of agent sessions it runs. The deployed engineer needs an outcome story that connects a changed engineering process to the business.

The chain might be: AI code review and testing reduce escaped defects; fewer defects reduce incident time; better reliability improves customer experience and retention. Or a migration completes sooner; the company retires infrastructure cost and unblocks a new product. Each link requires measurement and uncertainty. The agent platform does not deserve credit for every downstream event, but the organization should be able to explain why the automated workflow matters.

This business connection is another reason Factory's deployed role combines engineering knowledge with executive communication. A codebase-readiness score is useful to an engineering team. A transformation roadmap needs to connect that score to risk, cost, delivery, and strategic timing.

The Epcot warning

Reyes used Walt Disney's original idea for EPCOT as a warning about pilots. EPCOT was conceived as an Experimental Prototype Community of Tomorrow, a model city whose transportation and urban design could influence other cities. It ultimately became a theme park.

A deployed team wants to build a credible “codebase of the future” that proves a more autonomous operating model is possible. If the example is too modest, no one learns what could change. If it depends on exceptional staff, special permissions, a perfectly prepared repository, and endless vendor attention, employees elsewhere see a theme park: impressive, expensive, and unrelated to ordinary work.

The demonstration must be advanced enough to make the future tangible and ordinary enough to copy.

That means selecting an initial environment with real constraints, documenting every special condition, and planning how the customer's own team will scale it. The deployed engineer should leave behind a model, not a monument. A useful pilot teaches people how to reproduce the readiness work, workflow design, controls, and evaluation elsewhere.

Where autonomy arrives first

Reyes noted that autonomy may vary more by codebase than by company. A constrained internal tool with strong tests and a clear owner may become highly autonomous before a central product with complex visual behavior and many external dependencies. Asking which company reaches “100 percent autonomy” first is therefore less useful than asking which workflow has the strongest verification envelope.

This also prevents autonomy from becoming a slogan. Teams can report, for each workflow, how often the agent completes without interruption, which actions remain human, where validation is weak, what failure costs, and what investment would expand the boundary.

The deployed engineer at Factory is thus neither a consultant who completes every migration nor a salesperson demonstrating a coding agent. The role helps the customer build the conditions under which agents can do durable work, turns those conditions into a repeatable model, and feeds the hardest gaps back into the platform.

Questions for reflection

1. Which missing validator currently forces a human to stop an otherwise autonomous workflow?
2. Can your long-running task state a verifiable definition of completion?
3. What special condition makes your best pilot look like a theme park to the rest of the organization?

What the Speakers Agree On—and Where They Disagree

The eight talks do not produce a standard FDE job description. They produce something more useful: a set of recurring problems and competing organizational answers.

Sierra traces the role as a history of accumulated responsibilities. Anthropic supplies an economic test for whether it belongs in a company. Kepler treats field work as product discovery. Varick models the implicit operation of a business. Ramp pairs scope discipline with agent leverage. Decagon builds a funnel from custom work to self-service. Cognition turns customer production into an evaluation system. Factory asks how validation raises the ceiling of autonomy.

Their common ground is substantial.

Consensus 1: the real problem is not in the requirements document

Every speaker, in a different way, distrusts the request at face value.

Kepler's forty-seven-page dashboard became one operational alert. Ramp built Android support for a customer that allowed only iPhones. Varick finds the true workflow in exception paths rather than the documented golden path. Decagon examines historical support data before recommending which intents to automate. Cognition studies the entire software lifecycle instead of assuming code generation is the bottleneck.

The customer is not being dishonest. Requirements are produced from the perspective available to the author. A manager describes the desired artifact. A user describes a workaround as if it were the job. A sales team compresses a complex need into the feature most likely to receive attention. FDE begins by recovering the causal chain underneath the statement.

This is why presence has value. Close field contact increases the density of facts: repeated actions, contradictory definitions, failure recovery, and the small control that a proposed automation would accidentally remove.

Consensus 2: execution is getting cheaper; judgment is not

All eight talks assume that models and agents reduce the cost of producing code and other knowledge work. None concludes that enterprise deployment therefore becomes trivial.

When execution is expensive, companies are forced to choose carefully because every feature consumes scarce engineering time. When execution becomes cheap, teams can avoid the hard decision by building several versions. The cost reappears in maintenance, inconsistency, evaluation, and product drift.

Ramp calls the antidote scoping. Decagon calls it restraint. Kepler asks what product leverage the field work creates. Factory asks whether completion can be verified. The scarce skill is not merely deciding how to implement a request. It is deciding what deserves to exist, under which boundary, and with what evidence.

Consensus 3: one-off work has to return to the product

Every mature FDE model contains an upstream path.

The artifact may return as a connector, workflow primitive, ontology object, configuration schema, evaluation case, deployment package, self-service surface, or documentation. The precise form varies. The principle does not: if the same class of problem continues to require the same class of field labor, the platform is not learning fast enough.

Productization does not mean erasing customer differences. It means identifying which dimensions vary and giving those differences a maintained place in the system. The reusable asset may be a way to express local policy safely rather than a universal policy.

The return path also protects the business model. Without it, revenue scales with expert hours and customer-specific debt. With it, field work can increase the power of the product and reduce the marginal cost of later deployments.

Consensus 4: production adoption is the definition of done

The talks consistently distinguish a working prototype from a deployed operation.

A production system uses real data and authority. It has monitoring, audit, escalation, and support. Users rely on it during normal work. Failures are visible. The result can be measured over time. A beautiful demonstration may be a step toward this standard, but it is not a substitute.

Cognition makes the distinction through accepted pull requests and delivery outcomes. Factory makes it through verification and feedback loops. Sierra connects it to continuous customer accountability. Decagon insists on a narrow production slice before expansion.

The practical implication is that “time to first demo” and “time to value” are different metrics. The first measures the speed of construction. The second includes everything the organization must do before useful work changes.

Consensus 5: FDE cannot remain a heroic craft

The talks celebrate capable generalists but repeatedly design against dependence on them.

Anthropic recommends pairing and knowledge transfer. Varick builds an assistant to manage engagement context. Ramp automates intake and specification work. Decagon separates specialties and pushes configuration toward self-service. Sierra describes a mature organization as one that moves repeated responsibilities into the appropriate team and platform. Factory wants its product to self-assemble at enterprise scale.

The hero phase may be unavoidable at the beginning. A small team discovers the problem by doing everything. The danger is treating heroism as culture rather than temporary evidence of missing systems.

The question for every repeated intervention is: what would allow a competent person who was not present at the original deployment to understand, operate, and improve this safely?

Four productive disagreements

The differences among the speakers are not errors to reconcile. They reveal choices a company must make.

Disagreement 1: is FDE a durable role or a transitional stage?

Sierra suggests the title may dissolve as product engineers become more customer-facing and agent engineers take on deployment responsibility. Factory expects human work to move upward into designing the system that builds software. Varick explicitly imagines an agent handling more of the FDE's context and maintenance work.

Yet every transition creates new edge conditions. As low-level integration becomes self-service, customers attempt more complex workflows. As agents write code, verification and operating-model design become more important. The durable element may not be the job title, but there is little evidence that the need for boundary-crossing accountability disappears.

A sensible organization plans for both. It creates a role when concentrated ownership is useful, while steadily moving repetitive tasks out of that role.

Disagreement 2: does FDE belong to engineering or go-to-market?

Kepler strongly frames FDE as product strategy, especially for an early-stage company still discovering what to build. Decagon keeps the engineering bar and reporting relationship close to product engineering. Cognition says everyone is effectively go-to-market because customer success is the shared objective. Anthropic describes FDE historically as a go-to-market motion that turns a platform into outcomes.

The disagreement is partly about company stage. Early on, the biggest risk is building the wrong product; FDE should maximize product learning. With a mature platform, the risk shifts toward delivering and expanding value efficiently; the motion becomes central to go-to-market.

Where the team sits matters less than whether it has authority and obligations in both directions. If it reports to sales and cannot change the product, it becomes custom implementation. If it reports to engineering and has no commercial or adoption accountability, it becomes a remote product team with unusually noisy inputs.

Disagreement 3: ship the quick fix or wait for the general system?

Kepler encourages solving a problem in a day to earn trust, then warns that every temporary hack enters production. Ramp searches for workarounds before building the requested integration. Decagon emphasizes restraint and shared architecture. All recognize a tradeoff between time to value and future debt.

The decision should be explicit rather than ideological. A quick fix is appropriate when it is reversible, bounded, observable, and valuable as learning. A general solution is necessary when failure is expensive, the asset will become central, several customers already need it, or a one-off would bypass essential controls.

The team should state which mode it is using:

- **Probe:** disposable work to answer a question; no production dependency.
- **Bridge:** a temporary production solution with an owner, monitoring, and retirement date.
- **Primitive:** maintained shared capability intended for repeated use.
- **Product:** a supported user-facing capability with a stable contract.

Many disasters begin when a probe silently becomes a product.

Disagreement 4: is AI a tool for the FDE, or the future FDE?

Ramp describes agents replacing stages of the field pipeline. Varick is building an agent that may maintain workflows. Palantir's AI FDE product performs work associated with field engineers. Cognition and Factory use agents to execute increasingly long and complex engineering tasks.

At the same time, the speakers assign the human the most context-sensitive work: observe the organization, define the outcome, decide scope, negotiate control, judge product fit, and build the verification environment. AI can take more of those tasks only after the relevant context and standard have been made machine-readable.

The future is therefore recursive. FDEs build systems that automate parts of FDE; the work saved is used to model the next layer of ambiguity. The role moves from performing every action to designing the operating conditions under which actions can be trusted.

A comparison of the eight approaches

Lesson	Primary bottleneck	Role of the FDE	What must compound
Sierra	Fragmented accountability	Hold the customer thread across layers	Responsibility moves into teams and platform without disappearing
Anthropic	Complex product sold to a nontechnical buyer	Assemble outcomes on shared primitives	The platform expands and key-person risk falls
Kepler	Wrong problem definition	Observe, redefine, ship, and steer product	Field facts become ontology and product leverage
Varick	Implicit, exception-heavy operations	Model and redesign the workflow around AI	Company context becomes machine-readable
Ramp	Unbounded demand and manual field work	Scope continuously and automate the pipeline	Rubrics, context, and agent workflows improve
Decagon	Hypergrowth customization	Configure the agent and feed gaps upstream	Custom work becomes self-service; vertical knowledge accumulates
Cognition	Activity without delivery value	Map agents to high-value workflows and field evals	Production failures de-risk the roadmap
Factory	Weak verification limits autonomy	Build readiness, workflows, and outcome models	Validation loops and governance raise autonomy

The table makes the shared operating system visible. Enter the field. Establish the truth. Decide what result matters. Build the shortest responsible production loop. Observe failure. Convert the repeated structure into product and evaluation. Transfer ownership. Then begin again at a higher level.

Building an FDE Operating System

Forward deployment fails when it is treated as a collection of unusually capable people rather than an operating system. The following nine gates translate the eight lessons into a repeatable engagement. A project does not move forward merely because a meeting occurred or code exists. It moves when the evidence required by the gate is present.

Gate 1: Choose a customer worth deploying deeply

FDE attention is expensive and strategically revealing. Do not allocate it only according to contract size or executive volume.

Score a candidate on six dimensions:

1. **Business value.** If the workflow improves, does it change cost, revenue, risk, or strategic speed?
2. **Problem access.** Will the team receive users, data, systems, and decision-makers—or only a list of requests?
3. **Production intent.** Is there a sponsor prepared to change real work, not merely run a lab exercise?
4. **Verifiability.** Can the result and important failures be observed within a useful time window?
5. **Platform relevance.** Will the engagement exercise capabilities central to the product?
6. **Learning leverage.** Is the problem likely to recur in a market the company wants to serve?

A large logo with no access can consume months and teach very little. A smaller customer with a clear workflow and committed owner may generate the platform primitive that unlocks an entire segment.

Create a short **deployment thesis** before committing the team: why this customer, why this workflow, what the company expects to learn, and which evidence would cause it to stop.

Gate 2: Build a stakeholder and accountability map

An enterprise buyer is not a single actor. Map at least the following roles:

- economic buyer;
- executive sponsor;
- process owner;
- daily user;
- data and system owner;
- security, privacy, legal, and compliance reviewers;
- support and incident owner;

- product and engineering owner on the vendor side;
- person authorized to accept the result.

For each, record the desired outcome, authority, concern, evidence they trust, and definition of failure. A security team may define success as bounded access and auditable action. A user may care about fewer interruptions. A sponsor may need a quarterly financial effect. These are not competing details to smooth over. They are constraints the deployment must satisfy together.

Assign one person or small team to maintain continuity of customer accountability. That owner is not required to execute every task, but must be able to answer: why did the customer buy, what is in production, who uses it, what is failing, and what has the product learned?

Important engagements should have at least two people capable of takeover. Use pairing, shared calls, code review, and a regular “bus-factor drill” in which the secondary owner explains the system and handles a real change.

Exit evidence: stakeholder map, named acceptance authority, escalation path, and backup owner.

Gate 3: Draw two workflows

Document the official process and observe the real one.

The **declared workflow** comes from policy, procedure, system diagrams, and management interviews. The **observed workflow** comes from shadowing users and tracing real cases, including those that fail.

For every step capture:

- trigger and input;
- person, agent, or system performing it;
- data read and written;
- system of record;
- decision rule or judgment;
- permissions and approvals;
- normal output;
- exception, retry, escalation, and rollback;
- waiting time and work time;
- evidence that the step completed.

Look for high-information signals: copy-and-paste between tools, repeated tab switches, private spreadsheets, manual reconciliation, a phone used to complete a desktop task, scheduled checks, and phrases such as “I have to” or “only Maria knows.”

Do not automatically remove every workaround. A spreadsheet may be a hidden control surface that lets a domain expert correct bad data. Replacing it without preserving that power can make the workflow faster and less safe.

Exit evidence: side-by-side maps, at least three traced real cases, and a list of exceptions ranked by frequency and cost.

Gate 4: Rewrite the request as a verifiable outcome

Convert the requested artifact into an action loop.

Instead of "build a dashboard," write "notify the dispatcher within five minutes when a shipment will miss the promised window, provide the evidence, and record the replacement decision." Instead of "deploy a coding agent," write "resolve dependency-update tickets in these repositories, produce pull requests that pass these checks, and reduce median queue age without increasing escaped defects."

Use a one-page outcome contract:

- **User:** who experiences the problem?
- **Trigger:** what starts the workflow?
- **Decision or action:** what changes in the world?
- **Scope:** which entities, channels, regions, repositories, or intentions are included?
- **Non-goals:** what is explicitly excluded?
- **Control:** where can a human approve, correct, stop, or reverse?
- **Success metric:** what observable change counts?
- **Quality and risk guardrails:** what must not deteriorate?
- **Attribution:** what evidence links the deployment to the result?
- **Time window:** when will the team judge it?

This contract is not a sales promise written before discovery. It is a shared engineering instrument that changes when evidence changes, with decisions recorded.

Exit evidence: signed-off outcome contract and baseline measurements.

Gate 5: Compress to the shortest complete loop

The first release should be narrow in coverage and complete in operation.

Prefer one channel, one user group, a small set of intents, or a bounded repository class. Do not achieve speed by removing logging, permissions, evaluation, ownership, or rollback. Those are what make the first slice production rather than demonstration.

Choose a slice with:

- real volume and real users;
- a feedback period short enough to learn;
- errors that are detectable and containable;
- an existing system of record the result can reach;
- a sponsor able to change the workflow;

- a plausible expansion path if it works.

Classify the implementation before building:

Mode	Purpose	Production use	Required lifecycle
Probe	Answer a technical or workflow question	No	Delete or archive after the question
Bridge	Deliver time-sensitive value while a durable path is built	Yes, bounded	Owner, monitoring, expiry, migration plan
Primitive	Reusable technical or workflow component	Yes	Product ownership, tests, versioning, documentation
Product	Supported user capability	Yes	Stable contract, operations, roadmap, deprecation policy

The mode determines engineering rigor and ownership. A probe should not quietly receive credentials and a schedule. A bridge should not survive past its review date without an explicit decision.

Exit evidence: slice definition, mode classification, owner, and expansion/retirement decision point.

Gate 6: Design the human–agent control surface

“Human in the loop” is too vague to be a design. Specify where and why the human participates.

Four patterns cover many workflows:

1. **Human initiates, agent executes.** Appropriate when the user chooses the task and the action is bounded.
2. **Agent proposes, human approves.** Appropriate when evidence can be assembled automatically but the action carries meaningful risk.
3. **Agent executes, human audits.** Appropriate for frequent, reversible, well-instrumented actions.
4. **Agent executes within policy and escalates exceptions.** Appropriate when the normal path is measurable and the exception boundary is explicit.

For each agent action define:

- allowed tools and objects;
- read and write permissions;
- amount, scope, or frequency limits;
- evidence displayed to the reviewer;
- timeout and retry behavior;
- uncertainty or exception threshold;
- audit record;
- rollback or compensation action;
- person who receives an escalation.

Design for refusal as well as execution. A production agent should know when required evidence is missing, permissions are insufficient, or the action falls outside policy. “Ask a human” must route to a real queue with an owner and response expectation.

Exit evidence: control matrix, permission model, escalation queue, and tested rollback.

Gate 7: Enter production deliberately

A prototype becomes production adoption when real users rely on it in the formal environment and the organization can operate it without extraordinary intervention.

Before launch verify:

- identity and least-privilege access;
- data lineage and authoritative sources;
- environment separation;
- representative evaluation set, including exceptions;
- monitoring for quality, latency, cost, and business state;
- trace and audit retention;
- incident severity and response ownership;
- rollback or safe shutdown;
- user training and communication;
- support and change-management process;
- measurement baseline and reporting cadence.

Run a game day. Inject a missing field, unavailable API, conflicting instruction, permission denial, duplicated event, and incorrect model output. Observe not only whether the software recovers but whether the right human learns what happened.

Production launch is not the moment measurement ends. It is when the highest-fidelity evaluation begins. Collect accepted outcomes, corrections, escalations, and abandoned work. Link them to agent traces and product versions.

Exit evidence: completed readiness review, game-day results, go/no-go owner, and first measurement report.

Gate 8: Route every field asset upstream

At a regular cadence, classify what the engagement produced.

- customer-specific configuration;
- reusable vertical pattern;
- platform primitive;
- missing connector;
- evaluation case;

- documentation or training gap;
- product defect;
- unsupported custom debt;
- operating-model insight.

Use a **second-customer test**. If another customer presented a similar need tomorrow, what would the team reuse? If the answer is only “the original FDE's memory,” nothing has compounded.

For each recurring asset, assign a destination and owner. Productization should include not only code but the control surface, migration path, evaluations, and support model. A connector is not reusable if every customer still needs a developer to discover the same permission rule.

Create an explicit exception for needs that will remain bespoke. Document why they should not become product, who maintains them, and how their cost is reflected commercially. Generalization is a strategy, not a moral obligation to put every field artifact into the core platform.

Exit evidence: asset ledger with owner, destination, priority, and status.

Gate 9: Measure whether the motion compounds

An FDE dashboard should contain customer outcomes, delivery health, product learning, and organizational resilience.

Customer outcome

- time to first production value;
- adoption by intended users;
- business metric and quality guardrails;
- unresolved exceptions and incident impact;
- expansion driven by proven value.

Delivery health

- lead time from request to scoped decision;
- lead time from decision to production;
- percentage of work that uses supported primitives;
- manual interventions per workflow;
- change-failure and rollback rate.

Product compounding

- recurring field issues converted into evaluations;
- custom steps converted to configurable or self-service capability;
- reuse rate of connectors, patterns, and playbooks;
- reduction in time or engineering effort for comparable deployments.

Organizational resilience

- number of engagements with a trained backup owner;
- time for a new person to take over;
- incidents requiring the original builder;
- customer context captured in shared systems;
- repeated manual work assigned to an automation or product owner.

Do not confuse activity with progress. Tokens, sessions, generated code, meetings, and documents explain what the team did. They do not establish that deployment, adoption, or product leverage improved.

The most revealing compounding metric is comparative: for the fifth deployment of a recognizable pattern, is time to value shorter, is quality at least as high, and does the customer need less vendor-specific heroism than the first?

Three team shapes

No single org chart fits every stage. Three patterns recur.

1. Embedded generalists

A small number of strong engineers own discovery, building, launch, and feedback for early customers. This maximizes learning and speed when the product is uncertain.

Risk: key-person dependency and unbounded custom work.
Countermeasure: pairing, an asset ledger, weekly product review, and strict engagement selection.

2. Paired field and platform teams

FDEs own customer outcomes while a dedicated platform group converts repeated gaps into primitives, tooling, and deployment packages. The groups share planning and evaluation.

Risk: field urgency and platform roadmap become separate queues.
Countermeasure: shared metrics, joint technical design, and product owners accountable for field reuse.

3. Specialized deployment system

At scale, roles divide among discovery strategists, agent or solution builders, deployment engineers, vertical experts, and enablement. A platform and internal agents support context, workflow construction, and maintenance.

Risk: the customer thread fragments into departments.
Countermeasure: a named accountability owner and a shared engagement model linking business result, system state, and product learning.

Teams often evolve through all three. Problems arise when an organization copies the late-stage specialization before it has learned the work, or preserves the founder-like generalist model after volume demands systems.

Hiring for the work

The ideal FDE is often described as a unicorn: senior software engineer, consultant, product manager, salesperson, teacher, and operator. That description is a sign that the company has not decided which capabilities belong to the individual and which belong to the team.

Start with the minimum: the candidate should meet the technical bar for the changes they will own and be trustworthy in direct customer work. Then select for the actual product and stage.

Strong signals include:

- ability to turn an ambiguous request into a testable problem;
- comfort observing users without rushing to prescribe;
- technical depth in a domain important to the product;
- willingness to challenge urgency and explain a tradeoff;
- production instincts about permissions, failure, and ownership;
- ability to distinguish a local exception from a reusable pattern;
- clear written communication and decision records;
- habit of automating or productizing repeated work;
- low ego when field evidence contradicts an internal design.

Use work samples that reproduce the job. Give the candidate a fictional customer request with incomplete context. Ask which questions they would ask, what first slice they would choose, what they would refuse to build, how they would instrument it, and what might return to product. Then introduce a new constraint and observe whether they defend the original plan or update it.

Do not over-index on performance in a polished meeting. The work includes long stretches of careful synthesis, debugging, documentation, and follow-through after the exciting discovery. A credible FDE closes loops.

Finally, hire a portfolio rather than eight copies of one archetype. A team can combine deep infrastructure engineers, product-minded builders, vertical experts, and strong customer operators, provided responsibility and handoffs remain explicit. Sierra's “vintages” are a useful prompt: which layer is this person unusually prepared to own, and which layers should the surrounding system supply?

Appendix A — Glossary

The field combines enterprise software, data platforms, AI systems, and organizational design. The definitions below describe how the terms are used in this book.

AIP / Artificial Intelligence Platform. Palantir's platform for connecting generative AI and other models to enterprise data, the ontology, tools, applications, and operational controls. An AIP Bootcamp is a concentrated workshop in which customer and Palantir teams build a working use case from real business material.

Agent. A software system in which a model can interpret a goal, maintain state, choose or call tools, and take multi-step action. The word is used inconsistently across the industry; a deployment should define exactly which decisions and actions the system controls.

Agent Builder. A person who configures and evaluates an agent using supported product surfaces rather than changing the core platform. The role may combine domain expertise, workflow design, prompt and policy work, testing, and customer collaboration.

Agent Harness. The runtime system around a model: context assembly, tools, memory, permissions, execution, traces, validation, model routing, and governance. The model reasons; the harness determines how that reasoning can affect the world.

Agent Readiness. The degree to which an environment gives agents the context, tools, deterministic feedback, permissions, and safe operating boundaries needed to complete work. In Factory's framing, dense validation loops are a major determinant of readiness.

Air-gapped deployment. A system operated without a connection to public networks, usually for security or regulatory reasons. Models, dependencies, updates, observability, and support all need alternative paths inside the controlled boundary.

API / Application Programming Interface. A defined way for software systems to exchange data or request actions. An API connection is not a complete integration until identity, permissions, object meaning, error handling, limits, and write-back behavior are designed.

Business Intelligence / BI. Tools and methods for querying, reporting, and visualizing organizational data. A BI dashboard can make a condition visible; an operational workflow also connects that condition to decisions and actions.

Cassandra / Keyspace. Apache Cassandra is a distributed NoSQL database. A keyspace is its top-level namespace for tables and replication. Kepler's Phoenix story describes a missing date causing the system to generate millions of time-bucketed keyspaces.

Change Management. The work required for people and organizations to adopt a new process: communication, training, responsibility design, phased rollout, feedback, and treatment of incentives and

legitimate concerns.

CI / Continuous Integration. The practice of frequently merging changes into a shared repository and automatically building and testing them so defects and conflicts become visible early.

CLI / Command-Line Interface. A text-based interface used to operate software. Coding agents may expose a CLI in addition to an editor, web interface, or asynchronous cloud environment.

COBOL / JCL. COBOL is a business programming language still common in banks, insurers, and government mainframes. Job Control Language describes how batch jobs run on IBM mainframes. They represent the long-lived systems and specialized knowledge common in enterprise modernization.

Continuity of Customer Accountability. The preserved thread between why the customer bought, how the system was built, whether users adopt it, what failures mean, and what must return to product. It can be owned by a team, but it must survive departmental handoffs.

CRM / Customer Relationship Management. Software that holds customer, sales, and service information. Salesforce is a prominent example. A CRM is often a system of record that an agent must read or update without bypassing its permissions and audit history.

Custom-to-self-serve. The deliberate conversion of repeated manual or engineering work into supported configuration that an agent builder, implementation team, or customer can perform with appropriate guardrails.

Design Partnership. A close relationship in which a vendor and early customer jointly discover and shape a product. FDE can preserve this intimacy at enterprise scale only if field learning becomes reusable product capability.

Deterministic Validation. A check that produces a clear, repeatable result for the same input: compilation, a unit test, a schema rule, a type check, or a policy assertion. Deterministic feedback helps an agent revise work over a long horizon.

DevEx / Developer Experience. The tools, environments, documentation, workflows, and “paved roads” that let developers work effectively. Strong DevEx often improves agent readiness because it makes setup and validation explicit.

DevOps. A culture and set of practices that make software development and operations jointly responsible for build, release, monitoring, and recovery. It is not a single product.

ERP / Enterprise Resource Planning. A system for core business records and processes such as finance, procurement, inventory, and operations. SAP S/4HANA, Oracle NetSuite, and Microsoft Dynamics are examples.

ETL / Extract, Transform, Load. A data pipeline pattern that retrieves data from sources, cleans or changes it, and loads it into another system such as a warehouse or operational platform.

Evaluation / Eval. A systematic test of model or agent performance on a defined task. Enterprise evals may include answer quality, tool use, permissions, latency, cost, escalation, and business outcome—not only a benchmark score.

FDE / Forward Deployed Engineer. A customer-facing technical role that brings a complex product into a real environment, owns a path to production outcome, and carries field evidence back into the product. The title does not imply a single standardized job.

Foundry. Palantir's data operations platform, combining data integration, ontology, analytics, application building, workflows, security, and governance. It is not a standalone database.

Generalization. The act of identifying a stable pattern underneath a customer's local request and returning it to a platform, interface, evaluation, or operating method. It preserves meaningful variation rather than forcing every customer into an identical solution.

Golden Path. The standard route through a workflow when expected inputs and conditions hold. AI pilots often succeed on the golden path and fail on exceptions, retries, and recovery.

GTM / Go-to-Market. The system through which a product is discovered, sold, deployed, adopted, renewed, and expanded. An FDE team may be an engineering organization, a GTM motion, or both.

Human in the Loop. A design in which a person reviews, approves, corrects, or takes over at defined points. It can be a permanent control for high-risk work, not merely a temporary stage before autonomy.

IDE / Integrated Development Environment. A software workspace that combines editing, navigation, execution, debugging, and related tools. AI coding products often integrate into or provide an IDE.

MCP / Model Context Protocol. An open protocol for exposing tools and data sources to AI applications in a more standardized way. MCP helps a system discover and call capabilities; it does not decide when an action is appropriate or safe.

Mission. Factory's term for a long-running, multi-agent task container designed around bounded, verifiable knowledge work. In this book it illustrates the general principle that long-horizon autonomy needs a testable end state.

Model Routing. Selection of a model for a task according to cost, latency, capability, policy, or other criteria. A model-independent harness can route different stages to different models.

Ontology. An operational model of a business's objects, relationships, and actions. It can define what a customer, account, warehouse, aircraft, or invoice is; how those objects relate; and which actions authorized users or agents may take.

Observability / Audit / Rollback. Observability shows what a system is doing. Audit records who or what acted, when, and on which evidence. Rollback reverses a change or restores safe state. Together they are basic controls for production agents.

Outcome-based Pricing. Charging for a defined result rather than a user seat or unit of consumption. It requires agreement on measurement, attribution, quality, and what happens when the result is not achieved.

Parquet / CSV. CSV is a simple text table format that people can easily open. Apache Parquet is a compressed columnar format efficient for analytical systems. Kepler's story shows how a technically superior format can remove a user's practical quality-control step.

Primitive. A reusable platform building block such as identity, a connector, workflow step, policy, evaluation, or interface component. FDE solutions should be assembled from maintained primitives rather

than independent foundations.

Productization. Turning a customer-specific solution or repeated field pattern into a reusable, maintained capability. It includes tests, controls, ownership, documentation, migration, and support—not only cleaning up code.

Production Adoption. Continued use by real users in the formal environment, producing a measurable workflow or business result. It implies permissions, monitoring, support, incident response, and ownership.

Pull Request / PR. A proposed set of code changes submitted for review and merge into a repository. Generated code is activity; an accepted, validated pull request is closer to organizational output.

Reinforcement Learning / RL. A training approach in which a system learns from reward signals generated by its actions. Varick described an RL environment for teaching an agent to use specialized tools to traverse a company workflow graph.

Rubric. An explicit set of criteria for judging an artifact. A specification rubric may require user, outcome, non-goals, permissions, failure behavior, and metrics. A rubric improves consistency but does not eliminate product judgment.

SaaS / Software as a Service. Software operated and updated by a provider and usually accessed over a network under a subscription or usage model.

Sandbox. An isolated environment in which software or an agent can run with limited access, reducing the effect of mistakes. A sandbox still needs representative dependencies and validation if results are meant to reach production.

SAST / Lint / Type Checking. Static application security testing scans code without running it for known risk patterns. Linting checks suspicious or inconsistent code. Type checking validates how values flow according to declared types. They provide relatively clear automated feedback to agents.

Scoping. The continuous work of clarifying the user, outcome, boundary, dependencies, urgency, non-goals, and success standard—and deleting implementation that does not serve the result.

Skywise. An aviation data platform launched by Airbus with Palantir. It integrates data about aircraft, fleets, maintenance, parts, and operations, and is an example of enabling a large customer ecosystem to build on Foundry.

Software Factory. An observable production loop that turns signals into plans, agent and human work, validation, deployment, monitoring, and new signals. The central idea is a managed system for producing software, not an absence of engineers.

SRE / Site Reliability Engineering. The application of software engineering, automation, and measurable service objectives to reliability, capacity, and recovery in production systems.

Stakeholder Map. A record of the people and groups affected by a deployment, including their goals, authority, concerns, evidence, and definition of success.

SWE-bench. A software-engineering benchmark built from real GitHub issues and corresponding repository fixes. It evaluates whether a model can produce patches that pass project tests, but it does not reproduce the full enterprise delivery environment.

System of Record. The authoritative application or database for a class of business state and audit history. Agents usually need to integrate with it rather than create an uncontrolled parallel record.

Time to Value. The elapsed time between the start of an engagement and the first verified business or workflow result. It is stricter than time to prototype.

Token / Token-maxxing. A token is a unit used to process and meter text or code in a model. Token-maxxing is a critical term for maximizing model use or generated output without verifying valuable results.

Trace / Trajectory. A record of the context, reasoning-relevant steps, tool calls, outputs, and decisions in an agent run. Traces support debugging, audit, evaluation, and improvement.

Workflow Agent. An agent that performs a multi-step business or engineering process, maintaining state and handling dependencies, exceptions, and human handoffs rather than invoking a single tool once.

Appendix B — Speakers and Talks

Speaker	Role listed for the 2026 event	Talk used in this book	Lesson
Natalie Meurer	Head of Agent Engineering, Sierra	*The Dirty Secret of Forward Deployed Engineering*	1
Kevin Bai	Member of Technical Staff, Anthropic	*Forward Deployed Engineering 101*	2
Vinoo Ganesh	CEO and Co-founder, Kepler	*How Forward Deployed Engineering Is Done at Kepler*	3
Vasuman Moza	Founder and CEO, Varick Agents	*AI Tools for Forward Deployed Engineering*	4
Leo Mehr	Director of Engineering, Ramp	*How Forward Deployed Engineering Is Done at Ramp*	5
Sunny Rekhi	CTO of Forward Deployed Engineering, Decagon	*How Forward Deployed Engineering Is Done at Decagon*	6
Jia Wu	Deployed Engineering Lead, Cognition	*How Forward Deployed Engineering Is Done at Cognition*	7
Eno Reyes	CTO and Co-founder, Factory	*How Forward Deployed Engineering Is Done at Factory*	8

The second half of the Varick talk also included an engineering leader identified in the transcript as JD, who explained the internal FDE agent. The supplied subtitle did not provide a full identity that could be verified with sufficient confidence. The 2026 FDE track included additional sessions; this book covers the eight transcripts supplied for the project.

Appendix C — Discussion Guide

The questions at the end of each chapter are open-ended. The following prompts help a team turn them into an operating review.

1. **Locate the bottleneck.** Classify the current problem as missing reality, weak judgment, engineering constraint, product gap, or organizational adoption. Do not improve the model before identifying the layer.
2. **Challenge urgency.** Record customer urgency, sales urgency, regulatory deadline, and technical dependency separately. Decide which one governs sequence.
3. **Find the shortest action loop.** Start from the business action and work backward to the information and agent behavior it needs.
4. **Test productization.** Compare the shape of requests across customers. Look for stable objects, actions, controls, and evaluations underneath different language.
5. **Place the human deliberately.** The higher the error cost and the weaker the automatic validation, the earlier human approval should occur. Frequent, reversible, well-validated steps can move toward autonomy.
6. **Audit the pilot.** List every special permission, manual data cleanup, star engineer, and exceptional budget that made it work. Give each a replacement before claiming repeatability.
7. **Separate activity from outcome.** Report tokens, sessions, and generated work as activity. Report adoption, accepted output, cycle time, error, and business change as outcomes.
8. **Test continuity.** Ask a second engineer to operate the deployment, explain why decisions were made, and handle one exception without the original owner.

Appendix D — Sources and Accuracy

Primary material

The main source is a set of eight English automatic-caption transcripts from talks presented in the Forward Deployed Engineering track at **AI Engineer World's Fair 2026**. The event schedule lists the World's Fair for June 29–July 2, 2026 at Moscone West in San Francisco; the FDE sessions took place in the Track 8 room. The supplied videos and captions were published through the AI Engineer YouTube channel in late July 2026.

This independent educational edition thanks AI Engineer World's Fair and the speakers for making the talks available. The event name and logo, and the underlying talks, videos, captions, company names, and product marks, belong to their respective owners. Their inclusion identifies and links the source; it does not imply sponsorship or endorsement.

Background and name verification

- AI Engineer World's Fair 2026 and the official schedule: event dates, sessions, speakers, and roles.
- OpenAI — Forward Deployed Engineer: a current public description spanning discovery, scoping, system design, build, rollout, production adoption, and evaluation-driven product feedback.
- Palantir — Architecture Center: Foundry, AIP, Apollo, and the role of the ontology across data, logic, action, and security.
- Palantir — AI FDE: the public product boundary for agent-assisted data, repository, and ontology work.
- Palantir — Getting Started: AIP Bootcamp context.
- Airbus — Skywise launch: Airbus and Palantir collaboration and the aviation-data scope.
- AWS — What is DevOps? and Amazon EC2 concepts: background for the early infrastructure vocabulary.
- Apache Cassandra data definition and Apache Parquet: keyspace and file-format background.
- Ramp — Forward Deployed Engineering: Leo Mehr's fuller account of the Ramp motion, scoping, and generalization.
- Sierra — Natalie Meurer and Sierra product: speaker and product background.
- Decagon — Redefining forward deployment: the shift from heroics and custom work toward systems design and compounding delivery.
- Kepler — About and Kepler blog: Vinoo Ganesh's background and Kepler's emphasis on traceable, deterministic financial data.
- Cognition — Introducing Devin and the Cognition blog: the initial Devin system and later enterprise context.

- Factory — From coding agents to software factories and Software Factory: the signal-to-deployment feedback loop, model independence, governance, and autonomy spectrum.
- Varick Agents and AI for enterprise finance: company positioning around enterprise workflows and systems of record.
- SWE-bench FAQ: how the benchmark uses real issues, fixes, and project tests.

How to read the numbers

Efficiency figures, team sizes, migration scales, autonomy ratios, and customer examples in the chapters generally come from the speakers' own presentations or company materials. They are preserved when they clarify the argument, but they should not be treated as independently replicated benchmarks. The text identifies them as estimates, recollections, anonymized examples, or company claims where appropriate.

Automatic captions contain misspellings, especially in company and product names. Names were corrected when the intended referent was clear and could be checked. Uncertain model versions, unsupported statistics, and the full identity of the second Varick speaker were not forced into the text.

Independent adaptation statement

This is not an official publication of AI Engineer, the event organizer, the speakers, or the companies discussed, and it has not been reviewed or endorsed by them. Spoken repetition, stage banter, and recruiting appeals were removed. The arguments, examples, and important qualifications were retained; public documentation was added to help readers understand technical and company context.

CONCLUSION — WHAT GETS DEPLOYED IS AN OPERATING MODEL

Forward deployed engineering sounds like a job for carrying technology into a customer environment. Across the eight talks, the more important thing being deployed is a different way for a software company and a customer to share responsibility.

The product team can no longer assume that a technically available capability will become an operational result. Sales cannot use “the customer needs it” as a substitute for evidence. Engineering cannot treat generated code as completed work. The customer is not a passive recipient of a finished package; its users, systems, controls, and vocabulary are part of the system being designed.

FDE is not a cure for every enterprise problem. It can become an excuse for unlimited customization, a prestige title for support work, or a machine for letting the largest account capture the roadmap. It can produce indispensable heroes and fragile customer relationships.

The answer is not to stay away from the field. It is to make field work verifiable, transferable, and cumulative.

Turn requests into outcomes. Put the smallest complete loop into production. Treat exceptions as evaluation data. Move repeated work into primitives and self-service. Make personal knowledge into shared organizational memory. Build validation before claiming autonomy. Give every temporary asset a lifecycle.

Then ask the question that exposes whether the motion is working:

> After each deployment, is the company better at understanding customers and better at building the product?

If the answer is yes, the customer site is no longer merely a delivery cost. It is the fastest place the product can learn.

Produced by **Gao Fei's Digital Twin**
WeChat: **rohanjojo**
X: @feigaobox
Source acknowledgement: **AI Engineer World**
