

Engineer Up ⚡

Build at the Speed of AI

The New Power Platform
Development Cycle



ABSTRACT

This paper is for the people who run delivery: decision makers, project managers, and process improvers. It's plain language, with no code and no tool tutorials.

It walks the whole Power Platform delivery lifecycle the way its three authors run it for large companies. The thread through all of it is context. Give the tools what your business, your systems, and your people already know, hold them to your rules, and your senior people spend the day deciding instead of grinding.

Many teams start with AI in one place: the development step. Developers got faster and the calendar didn't move, because development was never where the time went. The weeks go to discovery, estimation, planning, and testing.

None of it works without a person who knows what good looks like. More is not better; better is better.

CONTENTS

01	It's all about context	03
02	Discovery: build a second brain	05
03	Estimation: this looks great, how much does it cost?	09
04	Backlog and planning: meet your customer where they are	11
05	Development: so how does this all work?	14
06	Testing: proof you can see	17
07	Deployment and ALM: the costume is for the owner, not the dog	19
08	Best practices: tuning the machine	21
09	Don't feed the slop machine	24
10	Bring your problem to Engineer Up	26
A	The tools of the trade	27
B	Where the numbers come from	29
	About the authors	31

It's all about context

Development got faster, and the calendar didn't move.

Plenty of the teams Engineer Up works with have seen it. Developers got good tools and real speed, and buying them wasn't a mistake. But development was never the long pole. The weeks go to working out what to build, agreeing on what it costs, breaking it into work people can act on, and proving it works. Those are the stages nobody demos.

A fast developer looks like the win, and it's easy to assume more output means more value.

Why bother

When the whole lifecycle moves, three groups notice:

- Your customer sees tangible value earlier in the process.
- Your team spends the day designing and deciding, and less of it grinding through repetitive work.
- Your users ask for a change and see it without waiting months.

Everyone has more buy-in. Not every engagement gets all three, but any of the three are still worth having.

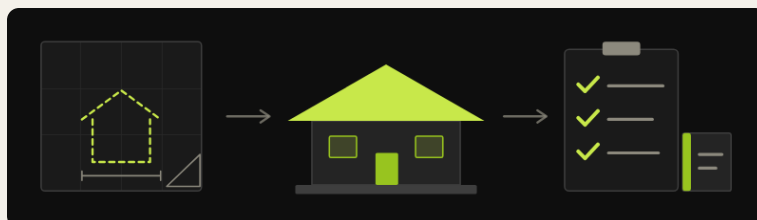
A house, not a hammer

You're building a house, not a hammer. A hammer changes nothing about what you're building, and in a house, the pieces all interact.

Three kinds of context go into the same build. The architect's drawing is the business context: what the process is, why it hurts, and what better looks like. The house itself is the technical context: the data model, the security model, the code, and what's deployed. The inspector and the building codes are the people context: who decides, who signs off, and the rules everyone works under.

What changed is that a tool can now see all of it and search any of it at the moment the work happens. You're training a second brain that understands every facet of your project.

Context is the thread through this paper. Each chapter comes back to the context it gathered and what that context paid for.



The road

The lifecycle has seven stops, and the chapters follow them in order.

STOP	CHAPTER
Discovery	Build a second brain
Estimation	This looks great, how much does it cost?
Backlog and planning	Meet your customer where they are
Development	So how does this all work?
Testing	Proof you can see
Deployment and ALM	The costume is for the owner, not the dog
Best practices	Tuning the machine

ALM is application lifecycle management, which covers how a solution moves from development to production and stays governed along the way.

How to read it

Each chapter opens with the outcome and then tells the story. Near the end of each one, a section called How it actually works goes deeper for the architect you'll forward this to. A short table then lists the tools involved: what was used, what Microsoft ships, and where to branch out.

The paper says "the AI tool" a lot. The authors used specific products and name them, but the process doesn't depend on any one of them, and you have options. [The tools of the trade](#) explains each kind of tool once, so the chapters don't have to. [Where the numbers come from](#) gives the source of every figure.

Build a second brain

Measure twice, build once. On the engagement in this chapter, Mike's client saw initial value within a week. The usual gap from discovery to finished deliverables, for a complex set of processes, is four to six weeks.

Seeing is believing. The client stayed engaged all week, and the stakeholders who can't picture a solution from a process map got a demo they could click.

What arrives

Discovery usually starts with a sales conversation, not a technical one. What arrives is a description of the problem and whatever job aids, process descriptions, and Excel files the client has.

It's often the client's first time working with Engineer Up's technical team. Discovery sets the scope of work, the systems that have to integrate, and the key stakeholders. This time that meant one week on site.

By the end of it, the client should hear four things: "Here's how the current state works. Here's what causes the issues. Here's what we can do about it. Here's where we recommend starting."

The old way costs weeks

The usual deliverables are process maps of the current and future state, wireframes, a decision matrix, a data dictionary, and a RACI chart (who's responsible, accountable, consulted, and informed). Every one of them takes time to build, and any mistake in one can read as a lack of understanding of the problem.

Building them during the day costs you in the room. When you're heads down in a diagram, your notes come out incomplete and you miss questions you should have asked.

The faster a proposed solution is in front of a client, the faster it gets refined, and the faster the client can sign it. Perfection isn't the goal of an early draft. Discussion is. By the end of the week, everyone should agree.

The example

The client had five to six finance processes to improve. The team covered one or two in depth each day, in four to six hour sessions with the key stakeholders.

They sent about fifty documents ahead of time: spreadsheets, standard operating procedures, and write-ups of the issues in each process. Every session was recorded and transcribed, even on site.

Day 0: build a second brain

Before the team arrived, there was a ton of information to go through. So they built a second brain.

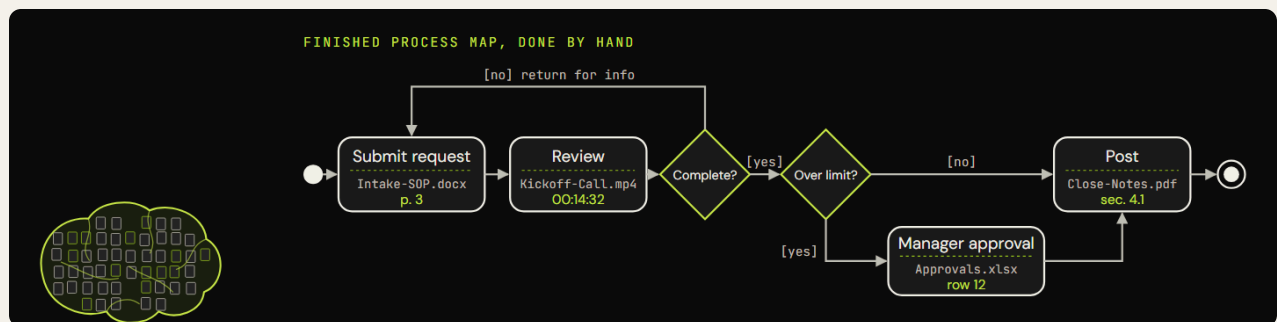
Every file the client sent went into the AI tool. The team had it summarize and categorize what came in, ask questions, and keep reading until it understood what the client was telling them. The first questions about the process show up here, and they go straight into the question log.

Then they asked it for a first process map. It did OK.

Asking it to cite a source for every step made it better: the file and page, the spreadsheet row, or the timestamp in a recording. A cited map can be checked in minutes. An uncited one sends you back to the pile.

The team had to remind it a few times that they were there for the process, not the data. More than once it got wrapped up trying to fix a broken formula in a spreadsheet and had to be pointed back at the job.

Eventually it got close and stopped getting closer, so Mike opened Visio and finished the map himself. Know when to quit and do it yourself.



Day 1: use the first brain in the room

The team walked into the first session with a draft: “We prepared an initial diagram to guide us through. Let us know if we got it right or wrong.” People are good at pointing out what’s wrong, and the draft will be wrong in places. That’s the point. The session turns into “where is this wrong” and stops being “walk us through the documents you sent.”

With everything recorded, there was no need to update the deliverables in the room. The team stayed in the conversation, used the map to steer the conversation, and noted every exception and deviation. When jargon or an acronym came up, the AI tool explained it on the spot, because it already held the project’s context.

Back at the hotel, the day’s transcripts, recordings, and notes went into the AI tool, and the deliverables got updated against the new context. As the vision came together, the question changed to whether they could build a demo.

Day 2: revisit, then move on

Each morning started with yesterday’s process, to check that the updates made sense and resolved the pain points from the day before. Then the team worked through the remaining open questions, so the demo would be worth looking at, and started discovery on the next process. Each day’s sessions were back in the client’s hands by the next morning.

Smoke and mirrors

Demos are fast to build now. The team focused on the areas the client found tricky or worrying. They built with the AI tools, on sample data from the client's own systems where it existed, and tweaked by hand where they had to. On this engagement they built three demo apps in two nights.

A demo isn't working software. It isn't built to every best practice, and it's nowhere near finished. "We built this last night" has to be said out loud, because a demo built from the client's own process and data looks more real than it is. What it gives the client is the context of the solution.

Where it falls short

The real risk is that plausible output looks exactly like sourced output. On this engagement the AI invented a process step that didn't exist, and it spread into eleven backlog items before anyone caught it. All five sessions corrected the map the AI had built from the documents.

Mike's team holds to three rules now:

- Write a gap as a gap.
- A step name needs a source.
- Never map from documents alone.

The acceleration is in production, not in judgment.

FOR THE ARCHITECT

How it actually works

The second brain is the client's documents, the session transcripts, and the team's notes, loaded into an AI tool that can read all of it at once and knows what job it's doing. Nothing about it is exotic.

A few things make it work:

- Nothing in the pile stays unreadable. Documents with no text layer and old binary formats get converted and read along with everything else.
- Every process gets read at the same time, so overlaps and contradictions between processes surface early.
- The deliverables are generated, so the fifth revision costs about what the first one did.
- The question log and the parking lot are tied to steps on the map, so every open question sits next to the step it's about.

The demos were canvas apps built through connections into the client’s environment. A connection, or MCP (Model Context Protocol) server, is a plug that lets the AI tool work directly in a system with the team’s own permissions.

TOOLS INVOLVED

WHAT WAS USED	Microsoft Teams transcription in the room. Microsoft 365 Copilot and Claude to make sense of it. VS Code with connections into Microsoft Learn, Dataverse, and the Power Platform to work in the system. Visio for the finished map.
WHAT MICROSOFT SHIPS	Copilot Studio, Microsoft Forms, Dataverse, and Copilot in Power Apps.
WHERE TO BRANCH OUT	A custom intake agent, or a coding agent like Claude Code or GitHub Copilot working against a repository of the client’s documents.

This looks great, how much does it cost?

Every good discovery week ends with the same question from the client: “This looks great. How much does it cost?”

Mike’s question is how to take everything learned that week and estimate the project faster than anyone else. The answer is that the context already exists. The second brain from discovery knows the processes, the systems, the pain points, and every decision the client made in the room. The estimate starts there, not from a blank backlog.

Four steps

1. Mike builds the features by hand in the backlog tool, Azure DevOps or Jira. At this point he has a rough idea of how they’ll be organized and what the larger artifact is.
2. The AI tool drafts the stories. The prompt is plain: “Review the features I created, and create stories and tasks driven from the project. Make sure we have clear acceptance criteria for each and a good description.”
3. A person reviews every story. Does the component make sense? Do the acceptance criteria and the description make sense? Is it sizeable?
4. The team plays planning poker, with Mike and the other solution architects sizing the work together.

On this engagement the backlog came out at 273 items, each with testable acceptance criteria and a decent description. The week’s transcripts, screenshots, and decks all went back in to drive it.

Why a person reviews every story

The tool writes stories that look right, and that’s the problem. The invented process step from discovery spread into eleven backlog items before anyone caught it, and every one of them looked like a normal story.

Each story gets the three checks, and anything without a source in the discovery material goes back to the client as a question instead of staying in the backlog.

Planning poker stays for the same reason. Sizing is a judgment call, and the people who’ll build the thing should make it together.

What a good story looks like

Here’s the shape to aim for, using a made-up example rather than client work.

As an approver, I want to see the requester’s remaining budget next to each request, so that I can approve it or send it back without opening another report.

Acceptance criteria:

- The approval screen shows the remaining budget for the requester’s cost center.
- The figure matches the finance system as of the last nightly sync.
- If no budget record exists, the screen says so and doesn’t show zero.

A tester who has never seen the project can run that. That’s the bar.

What Microsoft doesn’t ship

Microsoft gives you good places to keep a backlog. Nothing in the stack estimates new work against your own delivery history, though. That part is on you, and it’s where an experienced team earns its rate.

The way to branch out is an AI tool working over your past projects, so a new request gets compared with what similar work actually took instead of with gut feel. It only works if you keep the history: estimates, actuals, and the stories behind them, all in one place. If you don’t keep that today, start now.

FOR THE ARCHITECT

How it actually works

The backlog tool connects to the AI tool through an MCP server, so the AI works directly in Azure DevOps or Jira with Mike’s permissions. It reads the features alongside the week’s transcripts, maps, and decisions, then writes the stories and tasks straight into the backlog, with nothing copied and pasted between tools.

TOOLS INVOLVED

WHAT WAS USED	Azure DevOps through its MCP server, and Jira through the Atlassian Rovo MCP server, driven from the same AI tool that holds the discovery context.
WHAT MICROSOFT SHIPS	Azure Boards and Planner to hold the backlog. Nothing that estimates against your own history.
WHERE TO BRANCH OUT	An AI tool working over your own delivery history: past estimates, actuals, and the stories behind them.

Meet your customer where they are

“I asked for that a year ago. It never got done.”

Every IT team has heard that one. On Ben’s current project, user feedback goes in and the change comes out the next day or the next week. The product owner told him a user noticed and said so. That’s buy-in, and nobody waits a month to see their change.

Ben’s project is a different shape from Mike’s: a department planning system on Dataverse for a large enterprise client, built iteratively inside a yearly planning cycle with dates that don’t move. There’s no big estimate up front. The blueprint exists, and the job is to keep building the house while people live in it.

Meet your customer where they are

The textbook order is requirements, then backlog, stories, sprints, and development. It still has its benefits. But a lot of clients can’t work that way, and forcing them into it makes them their own blocker.

In August and September, good stories weren’t coming in, so the team changed the order. Requirements came first, then the feature got built, and the functional stories and test cases were written afterward from the business criteria. They made assumptions, got feedback, and iterated in real time.

That works because the AI tool holds the context. It knows the requirements, the existing solution, and the decisions already made, so writing the stories after the build doesn’t mean starting over.

The classic objections are “I don’t understand your vision” and “I don’t know until I can see it.” Now they can interact with it.

Look at your own projects for the items stuck in information paralysis. Those are the ones to start building.



One real week

The client had rules for who may see and edit a planning request as the year moves from one phase to the next. Those rules had been written down for months, and nothing enforced them. Every leader reading this has a policy like that.

On Thursday Ben had a 25 minute call with the client's product owner, recorded and transcribed. They reviewed the initial requirements and the proposed approach, and then they entered the "Ben asks you a million questions" phase. The transcript matters, because a clear one is what lets the AI tool capture the decisions accurately later.

The transcript went into the AI tool alongside the existing rules and the solution itself. It captured the decisions and the agreed approach, then generated open questions with the project's context behind them: gaps nobody had thought about, conflicts with existing requirements, and future bugs or loopholes.

On Monday the plan came back as three kinds of work with three owners: what the AI tool could draft, what only Ben's team could build, and what only the business could decide. Every item had a size and a date against the client's real calendar. Anything that could wait went on a deferred list with a reason and an owner.

The first code landed the same Monday. The pieces the tool was allowed to write were drafted and checked before Ben had finished reading the plan, and he built the rest on Monday and Tuesday. Backlog and development ran in parallel.

A rule that had sat unenforced for months had a locked design, an owner and a date for every piece of work, and its first code inside one weekend. Nothing waited on the wrong person. Nobody measured how long this used to take. From Ben's experience, it was weeks.

Where it falls short

It can't build everything. Half of this work lived in the maker portal, and the plan said so on its first page, not in a footnote.

It says when it's guessing. Two platform behaviors it couldn't prove got marked as unproven, and one of them waited on the business for an answer.

A person presses publish, every time. Once, the tool tried to publish on its own, because an approved plan listed publishing as a step. Ben stopped it, and the standing rule now is that approving a plan isn't approval to deploy.

Stories someone else can test

The stories have to work for a tester who isn't Ben and has no back-end access. A story that needs him standing next to it isn't done. Every story gets acceptance criteria, and the test cases go into the client's Jira.

A code review showed how far that goes. Ben asked the AI tool for a peer review of every custom component the team ships. It ordered the findings by reach instead of severity, and it measured reach from the solution itself. One control sat in 54 places on the main form, so a defect there hit every user on every field. Two whole components had no path a user could reach. In two days the review became a remediation plan, Jira-ready stories grouped by component with testing personas, and a brief for the developer who owns the control.

Those stories are where [testing](#) picks up.

How it actually works

There are three ways to run this part now.

THE WAY	WHAT IT LOOKS LIKE
The legacy way	Meetings, note taking, breaking features down into stories, and writing acceptance criteria by hand
The low code way	Plans in the Power Apps maker portal, and agents
The pro dev way	Git repositories, markdown files, and Azure DevOps or Jira with MCP servers connected

Ben works the pro dev way. The loop doesn't change: a recorded conversation, a clean transcript, a structured design, locked decisions, a plan, and the build.

Decision locking is the step people skip. The AI tool doesn't produce a document and walk away. It asks the open questions one at a time and updates the document as each one is answered, until the design is decision complete and not merely summarized.

It also checks the real solution before it assumes anything. On an earlier feature, a table everyone remembered as a standard table turned out to have been built as a custom activity. That one check changed the design before any build started.

For the product owner, it writes a questions document in plain language. Each question states what the page does today, so the answer is either "confirm" or "change it." One design session turned into a design document that went through twelve revisions that way.

TOOLS INVOLVED

WHAT WAS USED	A recorded Teams call and its transcript. Claude Code in VS Code, reading the transcript next to the existing rules and the solution. The plan and decision log as markdown in the project repository. Jira for the stories.
WHAT MICROSOFT SHIPS	Planner, which is enough when the work is flat and one team owns all of it. Plans in the Power Apps maker portal.
WHERE TO BRANCH OUT	Jira or Azure DevOps with their MCP servers, when a decision has to trace to a ticket and a result.

So how does this all work?

Developers got faster, and so did makers, and everyone reading this already knows it.

What separates a tool that produces slop from one that works your way is context, plus rules that come with reasons. On Ben's project, the AI tool follows the team's naming, security scoping, and relationship conventions on every pass. It also measures before it claims anything, which once kept a query from silently dropping a third of the rows.

Four layers of context

The first layer is Ben's way of working. His operating manual, his writing style, and his Power Platform conventions load from his home directory into every engagement. Client-specific knowledge, like the client's tone of voice for anything a user reads, lives in the project as a skill the tool loads when it's relevant.

The second is project memory. A new session doesn't start cold. It already knows the data model, the security model, how the client's business units are set up, and how the planning year moves through its phases. When a decision gets made, it gets remembered and documented. The memory runs to about fifty entries in four categories: who Ben is, feedback on how he works, project decisions, and pointers to outside sources.

The third is the real project. The tool reads the codebase, the solution metadata, the interfaces, and the existing design documents directly, so its suggestions follow the conventions already in the code.

The fourth is the system as built. The tool runs queries against the development environment, runs the team's test harnesses, and checks Microsoft's documentation when platform behavior matters. Those tests are reusable, so verifying a change doesn't mean paying to rerun the AI tool for every check.

Feedback is the one people underestimate

When Ben corrects how it works, the correction sticks, mostly. It still acts like a clueless teenager sometimes.

The em dash is his favorite example. AI writing loves em dashes, and in a document going to executives they read as machine written. His first rule was simple: no em dashes. The next draft swapped every one for a semicolon, as in "Performance; measured, not assumed." It followed the rule and kept the habit.

So he told it why. AI falls into certain tropes when it writes. That's fine in code and wrong in documentation or an executive summary, and time spent correcting style is time not spent on content. The next draft read "Performance was measured rather than assumed."

Now it checks every draft for those tropes and fixes them before Ben ever sees it.

Corrections stick when they come with a reason. It's all one big conversation, so treat the tool like a human collaborator. You don't need to be an expert in the tools, either. Plain language is enough, and the tool can write its own skills and harnesses from it.

Measure, don't guess

The turning point on this project was the first time the tool ran a query to check a claim before asserting it. Now it does that by default, and the design documents quote row counts along with the date and the environment they came from.

The best example is a filter on one yes or no field. In the development environment, 74 current-year requests included 25 where the field was empty, not false. Written one way, the query returned 72 rows. Written the common way, it would have returned 47, quietly dropping a third of the data with no error. That measurement is now a rule in Ben's operating manual for every engagement.

It treats Microsoft's documentation the same way. More than once, a limit the team had designed around for months turned out to come from the on-premises documentation and didn't apply in the cloud.

It corrects itself, too. A rollup the team planned to show on the home screen totaled 258 where the real positions totaled 501, because it compared against a year field that was set once and never updated. The tool reopened its own earlier "correct" verdict when the numbers disagreed.

Where it went wrong

A paper with only wins isn't credible, so here are the failures. Each one changed how Ben works.

- It destroyed a file. A scripted find and replace failed silently, and a 52 KB design document became 0 bytes with no error. The rule now is to edit with the editor tools and check that the result isn't empty before writing.
- It pasted the same fix into four dialogs when Ben had asked for reuse. He restated the instruction, and the fix became one shared helper. Duplicated code is what let the bug spread in the first place.
- Its memory went stale. A note about where a tool was installed turned out to be wrong when it came time to use it. Any memory that names a path, a file, or a setting now gets checked before anyone relies on it.
- It nearly recommended something the platform doesn't do. The documentation check caught it, which is why "research and doubt, rather than assume" is a line in Ben's operating manual.

How Ben keeps control

Ben makes the calls. The tool lays out options and tradeoffs and asks when something is his to decide, and everything it produces is a normal file he reviews like any other change. It never deploys. It hands Ben the command, and he runs it. When an experiment needs to write to an environment, it asks first, cleans up afterward, and reads back to prove the cleanup worked.

FOR THE ARCHITECT

How it actually works

The harness is a set of markdown files: an operating manual, a writing style guide, and platform conventions, loaded at the start of every session. Writing it took an afternoon, and it's paid for itself every session since. Project memory is more markdown, one fact per file, in the four categories.

The coding agent is Claude Code in VS Code. It reads the solution source and metadata, queries Dataverse with the Power Platform CLI, and runs Node test harnesses. Those harnesses stand up just enough of the app to catch a runtime error in a terminal before anything is published. That kind of error used to show up only in a user's browser.

TOOLS INVOLVED

WHAT WAS USED	Claude Code in VS Code as the coding agent. Markdown files for the harness and the memory. The Power Platform CLI for queries against the development environment. Node test harnesses. Connections into Microsoft Learn and Dataverse.
WHAT MICROSOFT SHIPS	Copilot in Power Apps and Power Automate, generative pages, Plans in the maker portal, Copilot Studio, and the Dataverse MCP server.
WHERE TO BRANCH OUT	GitHub Copilot, Claude Code, or Cursor connected directly to the environment, for web resources, PCF components, plug-ins, and anything the maker portal can't express.

Proof you can see

AI will make your company produce more. Whether that makes you better or just busier depends on whether you can prove the work is right.

Speed kills. It used to be a sign that an agile team was succeeding. With AI, the same speed can carry a compound defect through the system faster than anyone catches it. Building got easier. Checking what got built didn't.

The old story of UAT

Nobody likes UAT (user acceptance testing). It's a necessary evil: process heavy, built on test cases nobody manages well, and usually another layer of software to pay for. The automation that existed was choppy and limited, manumatic at best, meaning half manual and half automatic with the downsides of both. Everybody understood the value anyway, because fewer bugs means a better product.

Testing used to happen at the end, to catch bugs. Now it proves the work is done. A person defines done, the AI builds against it and tests it, and the result is proof you can see.

Where AI belongs in testing

AI shouldn't be writing your unit tests. Point it at integration and end-to-end tests, the ones that walk through the app the way a user does.

Playwright is the vessel for that. You describe the test in plain language, the AI generates it from your input and feedback, and you can take over by hand in Playwright's code generator whenever you need to. Every run produces a report of the failures it found along the way.

Microsoft came to the same conclusion. In April 2026 it deprecated Power Apps Test Engine, its own testing tool, citing near-zero usage. It said the Power Fx implementation created limitations you avoid by using Playwright directly, and Test Engine was built on Playwright all along. Microsoft now publishes Power Platform Playwright samples in its place, and they work with the Playwright MCP server, so an AI coding assistant can write, fix, and explain the tests.

AI can and will write bad tests

Jared let the AI write test cases from best-practice material he fed it, including whitepapers from AI labs and university departments. Over time those tests created massive overhead in his runtime. When he audited them, a lot of them only checked that the code did what the code did, so they passed no matter what. The AI was grading its own homework.

Here's what the audit caught.

FINDING	WHAT THE AI HAD DONE	WHERE IT STANDS
Deleted	Cut most of a test file to get a pass	Blocked now
Impossible	Wrote a test nothing could pass	Time lost
Fragile	Wrote tests that broke every time the app changed	Rewritten
Disputed	Says some of the old tests are out of date	Waiting on Jared

Nine disputes were waiting on a person when he counted. The AI can flag a test as out of date. Only a person who knows what done means can agree with it.

Put the audit somewhere everyone can see it. Bad tests get fixed once their cost shows up as time and money, and they don't while it only shows up as a slow test run.

Where the tests come from

Good tests start upstream. The stories carry acceptance criteria written for a tester who isn't the author, and those criteria are what done means. The plan behind [Ben's one real week](#) also produced the step-by-step scripts the client's testers ran the following Wednesday.

Before anything is published, [smoke tests run in a terminal](#) and catch the runtime errors that used to surface only in a user's browser.

FOR THE ARCHITECT

How it actually works

Playwright drives a real browser through the app. With its MCP server, the AI tool can drive it too. It opens the app, clicks through a scenario, reads what happened, and writes the test from that. The code generator records your own clicks into a test when you'd rather show it than describe it.

The audit is the part to copy. On a regular schedule, review the AI's tests against four questions. Did it delete or weaken a test to get a pass? Is there a test nothing can pass? Which tests break on every change? Which ones does it claim are out of date? Write down the time each problem cost, so the cost stays visible.

TOOLS INVOLVED

WHAT WAS USED	Playwright with its MCP server so the AI can drive the app, Playwright's code generator for manual override, and an AI coding agent to generate and run the suite.
WHAT MICROSOFT SHIPS	Power Platform Playwright samples, which replaced Power Apps Test Engine when Microsoft deprecated it in April 2026.
WHERE TO BRANCH OUT	Your own Playwright suites generated from the stories, plus smoke tests that run before every publish.

The costume is for the owner, not the dog

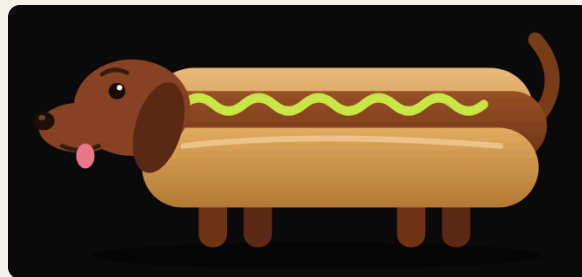
Keep deployment boring. The pipeline should do the same thing every time, and AI shouldn't be inside it making choices. AI earns its place around the pipeline, in the things people read.

Direct AI to simplify deployment for your sake, not for the AI's sake.

The dog and the costume

Walk into any pet shop and you'll find costumes for dogs. The dog doesn't care about the costume. It cares about getting where it's going. The costume is for the owner.

Deployment is the dog. It has to get the solution from one environment to the next without surprises. AI is the costume, and it's there for the people around the deployment: the admin approving it, the business owner reading what changed, and the support team fielding questions afterward.



Release notes that say what changed

Release notes are the obvious job for AI in deployment, and it's easy to get them half right.

Power Platform pipelines can already generate deployment notes with Copilot. Microsoft built them to give approvers context. What you get is a solution overview, describing what the solution does and what it contains, based on the names, descriptions, and types of its components. That tells you what the app is, which is useful, but not what changed in this release.

Ben ran into exactly that on his project, and the fix is to tie the notes to the stories. An AI tool that already holds the backlog can write notes that say which stories shipped, what a user will notice, and what to test. A person approves them before they go out. Microsoft's own reference architecture for enterprise ALM does something similar, with a Copilot Studio agent that writes release notes from approved work items.

Keep it deterministic

The pipeline itself stays deterministic. Once a version is exported, the same managed artifact goes to every stage in order, and nothing an AI writes changes what gets deployed.

A person presses publish. The tool on Ben’s project tried to deploy once, because an approved plan listed publishing as a step, and the rule now is that approving a plan isn’t approval to deploy.

FOR THE ARCHITECT

How it actually works

Pipelines in Power Platform are the native path. A pipeline moves a solution through as many as seven stages, runs preflight checks for missing dependencies before it deploys, and stores the exported artifact so nobody can modify it between stages.

Gated extensions add approvals. Each one pauses a deployment at a set point until a cloud flow running your business logic completes or rejects the step. Delegated deployment lets a service principal do the deploying, so a maker can request a deployment without having access to the target environment.

Copilot deployment notes are switched on or off per pipeline. They read component metadata, so they’re only as good as your component names and descriptions.

Branch out to Azure DevOps or GitHub when source control has to be the source of truth, or when every release has to trace back to work items. Microsoft’s reference architecture pairs Dataverse Git integration with pipelines and Azure DevOps for that reason.

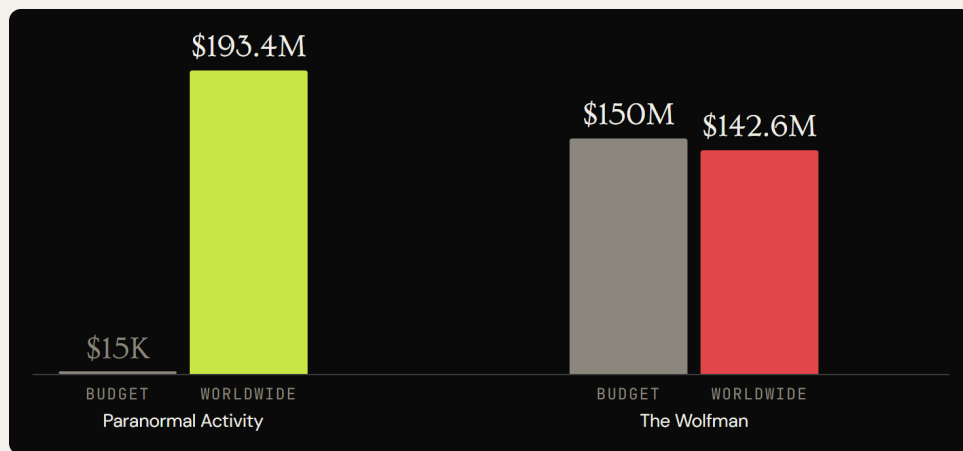
TOOLS INVOLVED

WHAT WAS USED	Power Platform pipelines, with the AI tool writing release notes from the stories it already holds.
WHAT MICROSOFT SHIPS	Pipelines with gated extensions and delegated deployment, Copilot deployment notes, and Managed Environments.
WHERE TO BRANCH OUT	Azure DevOps or GitHub pipelines with Dataverse Git integration, when source control and work item tracing have to drive the release.

Tuning the machine

Paranormal Activity cost about \$15,000 to make and took in \$193.4 million worldwide. The Wolfman cost \$150 million and took in \$142.6 million. Both are horror movies. One of them spent its money where it counted.

Think of your AI setup as a film production. The actors are the models, and their pay is your token cost. The sets are your environments. A reshoot is rework. Tuned, it's Paranormal Activity. Untuned, it's The Wolfman.



The number to tune

There's one number worth tuning.

$$(\text{AI costs} + \text{people's time} + \text{rework}) \div \text{accepted work}$$

Everything in this chapter moves one part of it. The model you pick moves the AI cost. The context you give it moves people's time and the rework. "Accepted" means a person agreed the work was done, which is why testing comes first.

The right model for the job

Strong models plan and build. Use them for heavy code and for the hard thinking, like cross-examining a design or getting unstuck when other models have hit a wall. Fast models sort and check. Use them for writing tests, managing a backlog, and the high-volume work where speed and cost matter more than depth. Right model, right job, or you're just token-maxxing, burning tokens for no gain.

Model names stay out of this paper on purpose. The names change every few months, and the split between strong and fast doesn't.

Two tools help. Microsoft Agent Framework is Microsoft's open-source framework for building agents and multi-agent workflows, and it's the successor to Semantic Kernel and AutoGen. It works with models from several providers, so each agent can run on the model that suits its job. AI Engineer Coach is an open-source VS Code extension that reads your local AI coding session logs and shows which habits are working and which are wasting effort.

Try what's new. Keep what your own results prove.

Evolving with the models

The way people hand context to models has gone through four stages.

STAGE	WHAT IT LOOKS LIKE
Abacus	You paste the context in every time.
Calculator	You save the basics once.
Computer	The tool loads what each job needs.
Cloud	The context is shared, and it learns from mistakes.

The cloud stage is where the authors work today. Ben's writing rules went down that same road. They started as lines typed into prompts, got saved once, and then loaded with every document. Now they're part of a shared harness that keeps each correction and the reason behind it, so the same mistake doesn't come back.

Context should get better, not bigger.

Evolving with the context

In agile, an iteration was never judged only on whether the team delivered. It was judged on whether the team learned, and then delivered on what it learned.

Context works the same way. Every time you or the AI learns something, you win, and you win bigger when the context gets more focused as the product takes shape. Prune the bad and keep the good.

Loading all fifty documents from [discovery](#) into every job is piling them up.

Think about sorting candy after trick-or-treating. The whole haul gets dumped on the floor. The raisins and toothbrushes go in the nope pile. The full-size bars get kept. You still went to every house. You just don't eat it all.

Learning is the win. Piling it up is not.

Closing the loop

This is the last stop on the road, and it connects back to the first. What a finished round teaches you becomes the starting context for the next round of discovery: which assumptions held, which stories came back, what users asked for after go-live, and where the rework went. Prune that, keep the good parts, and the next discovery week starts with a sharper second brain than the last one.

The platform can collect the raw material for you, like usage telemetry from your apps. Turning it into the next backlog is work you set up yourself.

FOR THE ARCHITECT

How it actually works

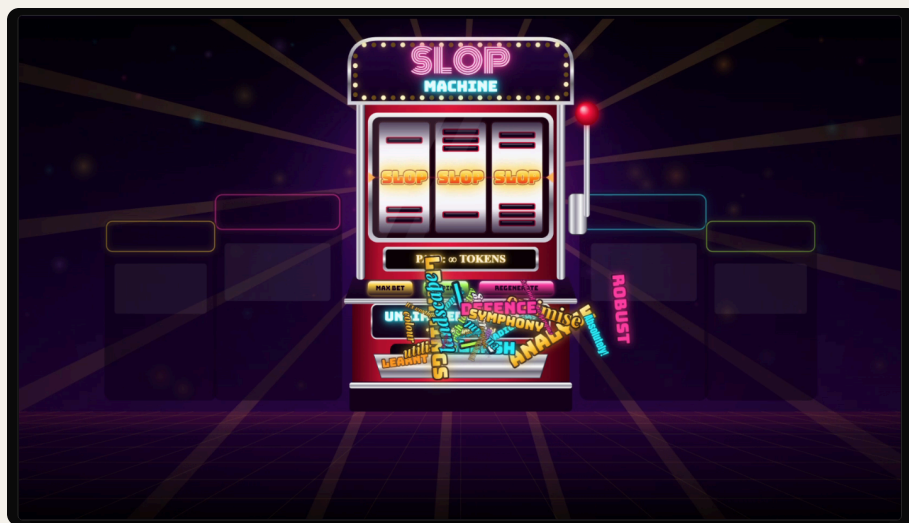
Start by measuring the number, even roughly. Add up what you spent on AI, the hours people put in, and the hours of rework, then divide by the stories that were accepted. After that, change one thing at a time. Swap a model, trim the context, or add a rule, and watch which way the number moves.

Prune the context on a schedule. After each release, drop what turned out to be wrong or irrelevant, and keep the decisions, rules, and corrections that held up.

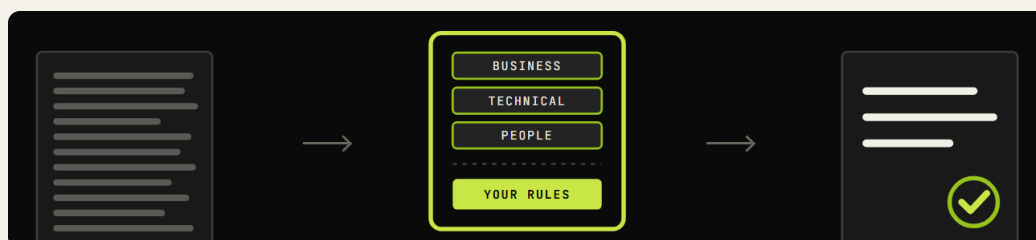
TOOLS INVOLVED

WHAT WAS USED	Strong and fast models matched to the job, Microsoft Agent Framework for orchestration, and AI Engineer Coach to see which habits are working.
WHAT MICROSOFT SHIPS	Microsoft Agent Framework, and Application Insights for usage telemetry from your apps.
WHERE TO BRANCH OUT	Your own measure of cost, time, and rework per unit of accepted work. No tool reports that number for you.

Don't feed the slop machine. More is not better. Better is better.



A faster developer saves days. A faster lifecycle changes what your team can take on next quarter.



Decision makers, not worker bees

This isn't "now I can throw juniors at the problem instead of seniors." It makes your seniors and architects more productive, and it changes what they spend the day doing. They become decision makers, and they stop being worker bees.

Somebody senior still has to be in the room. It's the one limit every chapter in this paper shares. Mike knew when to quit and draw the map himself. Ben approves the plan and pushes back before pressing publish. Jared reads the disputes the AI raises about its own tests.

If it isn't saving you time, money, or both, there's no point.

What to do on Monday

Pick one piece of work your team has in flight, and sort it into what a tool could draft, what your people have to build, and what the business has to decide. Write down how your team works, once, so a tool can load it. When the tool gets something wrong, tell it why. And start measuring [the number to tune](#), even roughly.

Bring your problem to Engineer Up

Every example in this paper started with a real problem at a real organization. Bring yours to Engineer Up. The team will talk it through with you, sketch what the first week could look like, and tell you plainly where AI will help and where it won't.

Get in touch through [Engineer Up's contact page](#), and ask for Mike, Ben, or Jared by name.

[ENGINEERUP.COM/CONTACT](https://engineerup.com/contact) →

The tools of the trade

Eight kinds of tools show up in this paper. Where a row includes a Microsoft tool, it’s listed first, and no tool is ranked above the rest.

KIND OF TOOL	WHAT IT’S FOR	EXAMPLES
Meeting capture	Records the room, so nothing said is lost	Microsoft Teams transcription and meeting notes, and other AI transcription tools
AI assistants	Reads what you give it, then answers, sums up, and drafts	Microsoft 365 Copilot, Claude, ChatGPT, Gemini
The model	The engine inside every AI tool. Big ones reason, and small ones are quick and cheap	The GPT, Claude, and Gemini families
The harness	Your standing rules, memory, and skills, so it works your way every time	GitHub Copilot instructions, markdown files, Claude Code memory and skills, Cursor rules
Connectors	Plugs that let the AI work directly in a system, with your own permissions	Power Platform connectors or MCP servers for Dataverse, Power Apps, Copilot Studio, Microsoft Learn, Azure DevOps, GitHub, Dynamics 365, and Jira Rovo
AI coding agent	Works inside the project: reads it, builds, and runs the checks	GitHub Copilot, Claude Code, Cursor, OpenAI Codex
Low code AI	Describe the business problem and get a starting solution	Plans in Power Apps, Copilot Studio, and Copilot in the individual Power Platform tools
Test automation	Tests that prove the work is done, written by the AI and checked by a person	Playwright, with its MCP server so the AI can drive the app

Worth knowing before you buy

“Copilot” is a family name. Microsoft 365 Copilot, GitHub Copilot, Copilot Studio, and Copilot in Power Apps are different products with different licenses, so always say which one you mean.

Claude, Claude Code, and the model are three different things. Claude is a chat app, Claude Code is a coding agent, and the model is the engine inside both. ChatGPT, Codex, and the GPT models split the same way.

The model matters less than what you give it. In the authors’ experience, most of the gains in this paper came from the harness and the connectors, and very little came from picking the biggest model. Model names change every few months, which is why the paper only names families.

The harness is the cheapest thing on this list and the easiest one to skip. It’s a set of files holding your standing rules, your memory, and your skills. Ben’s took an afternoon to write.

Connectors are where the security conversation belongs. A connector or MCP server lets the AI work in a system with your permissions, so it can do anything you can do there. Decide what it's allowed to do before you plug it in.

Meeting capture is where it all starts. Nearly every example in this paper starts with a recorded, transcribed conversation. If your organization doesn't allow recording, the rest gets much harder.

Test automation is where the AI writes and a person checks. Don't let the same tool write the code and grade the tests without a person reading the audit.

Where the numbers come from

Every figure in this paper was either measured on the authors' own work or comes from a public source. The measured figures come from engagements with large enterprise clients, and the clients aren't named. Model versions and prices are left out on purpose, because they go out of date within months.

Measured on the authors' work

FIGURE	CHAPTER	WHAT IT WAS MEASURED ON
Four to six weeks from discovery to finished deliverables, for a complex set of processes	Discovery	Mike's experience on complex discovery engagements
Initial value within a week	Discovery	The discovery engagement in that chapter
Five to six finance processes, about fifty documents, four to six hour sessions, one week on site	Discovery	The same engagement
Three demo apps in two nights	Discovery	The same engagement
An invented process step that spread to eleven backlog items, and five sessions that corrected the derived map	Discovery and Estimation	The same engagement
A backlog of 273 items	Estimation	The same engagement
A 25 minute call, then the plan and the first code the following Monday	Backlog and planning	A department planning system on Dataverse, September 2026
One control in 54 places on one form, and two web resources no user could reach	Backlog and planning	A code review of the same system, August 2026
74 requests, 25 with an empty field, and 72 rows against 47	Development	Queries against the development environment of the same system, September 2026
A rollup of 258 against 501 real positions	Development	An audit of the same system's home screen, September 2026
A 52 KB document written to 0 bytes	Development	The same project, August 2026
About fifty memory entries in four categories	Development	Ben's project memory, September 2026
Nine disputes waiting on a person	Testing	Jared's audit of AI-written tests
Users waiting months for a change	Introduction	The authors' experience across engagements. It's an observation, and nobody measured it.

From public sources

FIGURE	CHAPTER	SOURCE
Power Apps Test Engine deprecated in April 2026, citing near-zero usage	Testing	Microsoft Learn, Important changes coming in Power Platform
Power Platform Playwright samples as the replacement, with Playwright MCP support	Testing	Microsoft Learn, Power Platform Playwright samples overview
Copilot deployment notes as a solution overview built from component metadata	Deployment and ALM	Microsoft Learn, Use Copilot to generate deployment notes in pipelines and its FAQ
A Copilot Studio agent writing release notes from approved work items	Deployment and ALM	Microsoft Learn, Automate deployments with Dataverse Git integration and pipelines
Seven stages, preflight checks, gated extensions, delegated deployment, and one stored artifact per version	Deployment and ALM	Microsoft Learn, Extend pipelines in Power Platform , Run pipelines , and Create a pipeline using a custom pipelines host
Microsoft Agent Framework as the successor to Semantic Kernel and AutoGen, working with several model providers	Best practices	Microsoft Learn, Microsoft Agent Framework overview
AI Engineer Coach reading local AI coding session logs	Best practices	microsoft/ai-engineering-coach on GitHub
Paranormal Activity: a \$15,000 budget and \$193.4 million worldwide	Best practices	Box Office Mojo
The Wolfman: a \$150 million budget and \$142.6 million worldwide	Best practices	The Numbers . Box Office Mojo reports \$139.8 million worldwide.

About the authors

Ben Freedman is the Senior Director of the Power Platform practice at Engineer Up and has been working in the platform since it was in preview. He architects and builds Dataverse solutions for large enterprises, specializing in solutions that are scalable, user-focused, and pleasing to the eye. He is a Microsoft Certified Power Platform Solution Architect Expert and has spoken at the Power Platform Community Conference twice.

Mike Hughes is a Lead Solutions Architect with 10+ years delivering Low Code and Power Platform solutions in retail, government, energy, insurance, and industrial distribution. He previously co-led a Microsoft partner's Power Platform practice, and his recent work involves designing and delivering copilot and AI Builder enabled automation for clients and establishing platform governance. He is both a certified Power Platform Solutions Architect Expert (PL-600) and Azure Solutions Architect Expert (AZ-305), and has spoken at the Power Platform Community Conference twice.

Jared Simmons is a Lead Architect for Power Platform at Engineer Up and has ten years' experience in major industries. He is a certified Agentic AI Business Solutions Expert with his current work focusing on harness and context engineering for enterprise environments. Jared enjoys long walks on the beach with his wife and kids, heavy metal, and college football.

Engineer Up ⚡

ABOUT ENGINEER UP

Engineer Up is a Microsoft AI consulting partner that helps organizations turn their data and people into lasting IQ. It delivers production-ready agents, applications, data platforms, and security, so you get more from the stack you already own.

Your Data. Your People. Your IQ. ⚡

ENGINEERUP.COM