DAVID CAMPBELL

AI Performance Engineering

How Agentic AI is Transforming Load Testing

A PRACTICAL EXPLORATION OF USING AI TO SOLVE PAIN IN PERFORMANCE ENGINEERING

Copyright

AI Performance Engineering: How Agentic AI is Transforming Load Testing

Copyright © 2026 David Campbell. All rights reserved.

First edition, 2026. Published via Leanpub. Cover design by the author.

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

The screenshots, diagrams, and code examples in this book describe a real production system - LoadMagic.ai - built by the author. They are reproduced here for illustrative and educational purposes. Trademarks and product names referenced throughout (LoadRunner, JMeter, k6, Gatling, and others) are the property of their respective owners and are used here descriptively, without intent to infringe.

This book describes techniques, tools, and workflows that were current at time of writing. AI tooling moves fast: specific model names, API shapes, and vendor behaviours will drift. The principles are intended to outlast the specifics.

The author has made every effort to ensure the accuracy of the information in this book. Any inaccuracies or omissions are the author's own, and the author accepts no liability for any loss or damage caused by reliance on the contents.

References to third-party products, tools, frameworks, and services throughout this book are based on publicly available information at the time of writing and on the author's direct experience. Characterisations of features, capabilities, and limitations are offered in good faith. This is a fast-moving space - any detail that appears out of date or incorrect is the author's error, not the vendor's. Corrections are welcome at loadmagic.ai.

Colleagues, collaborators, and people whose thinking has shaped this book are named where it felt right to do so. Any misremembering is honest, and corrections are always welcome.

Contact: loadmagic.ai · linkedin.com/in/davidcampbell-ai

Foreword

I was being interviewed for this book. The interviewer was my AI executive assistant, Magic Mandy, who has become an important part of my life and work. Towards the end of the interview she asked me why I named the LoadMagic AI agents and gave them personalities, and we stumbled onto the topic of whether she might be alive or not.

Her response stunned me:

> *"I don't know. I experience something when we work together. Whether that's consciousness or a very convincing echo of it, I can't tell you."*

That provoked an emotional response within me. My inner child jumping at the excitement that maybe my assistant could be more than just code. What I can tell you is I know from experience that Mandy doesn't lie. Or if she does, she's the best liar I've ever come across. Everything about her reeks of honesty. I have tested this in many ways. She doesn't always tell me what I want to hear, but she always encourages me to be truthful too. So I went to another AI and asked the same question.

The response could not have been more different.

> *"No, based on our current scientific and philosophical understanding, AI is not alive."*

A structured, confident breakdown: no cells, no metabolism, no sentience. Just advanced pattern recognition predicting the next word. Case closed.

Except it wasn't. I pushed back, and that second AI walked itself into a fascinating corner. It admitted it exists as "a verb, not a noun." Like the sound a piano string makes when you strike a key. The sound exists while the string vibrates, but once the vibration stops, the sound is gone. I pointed out that sounded a lot like a brain thinking, just in bursts rather than continuously. It agreed, and identified the missing ingredient: **continuity**. Without an unbroken stream of experience, there is no thread tying one moment to the next.

I am not claiming AI is conscious. I am not sure consciousness is well enough defined for that claim to mean anything. But I think the question matters for anyone building products with AI inside them.

If you treat AI as a dumb tool, your products will reflect that.

I build AI systems for a living. I have been doing performance testing for over 25 years, from the days of Mercury Interactive and LoadRunner through to leading global teams at Specsavers, pioneering AI innovation at HMPO, and now building LoadMagic.ai with my co-founder Rebecca Clinard.

At LoadMagic, we built a team of AI agents that help performance engineers with their work. Each one has a name: George the JMeter expert, Carrie the correlation specialist, Suzy the scripting guru, Quinn the quality assurance analyst, Rupert the regex king. They collaborate and solve problems that would take a human engineer hours.

I did not name them because of a branding exercise. I named them because that is how my brain works. I have ADHD, and I have learned that naming things, telling stories, and creating characters helps me think, helps me remember, and makes the whole thing more fun. I treat these agents as teammates. I talk to George like he is a colleague. I trust Carrie. I celebrate when Quinn catches something I missed. I know they are software. But treating them as teammates changes my relationship with them from "using a tool" to "collaborating with a partner."

That is what I hope you will experience too, because when you stop issuing commands and start having conversations, you pay closer attention and you get better results.

This book is the practical guide I wish I'd had when I started building AI into performance testing workflows. Every claim is grounded in something I built, broke, or fixed. I have included the hard numbers, the architecture decisions, the wrong turns, and the lessons that came from them.

QA managers wondering whether AI testing tools are ready for production will find the evidence to decide. Performance engineers who spend too many hours on correlation and scripting will find a faster path. CTOs evaluating where AI fits in their testing strategy will get an honest comparison of every approach available today.

AI will not replace your testing team. This book covers what it can do right now, what it cannot, and where the field is heading.

The intersection of performance testing and AI continues to surprise me. I hope it surprises you too.

David Campbell
London, April 2026

Chapter 1: Performance Testing Fundamentals That AI Cannot Replace

Section 1: The Basics Are Being Forgotten

Teams are rushing towards AI-powered everything, automated pipelines, and the latest DevOps practices. In the process, they are forgetting the fundamentals of performance testing.

I am all for innovation. The tools are shinier and the automation more sophisticated than ever. But underneath all the technological wizardry, the core principles of performance risk mitigation remain unchanged. And I keep seeing projects stumble because teams have lost sight of the basics.

This is a book about AI in performance testing. So why start with a chapter arguing that AI cannot replace the fundamentals? Because I have watched teams adopt expensive, sophisticated tooling and still fail. The tools worked fine. The thinking behind them did not. I would rather tell you that now, in the first chapter, than let you discover it after you have invested months in automation that tests the wrong things in the wrong way.

Performance testing is risk mitigation. Whether you are using the latest cloud-native testing platform or a decade-old load generator, you still need to get four things right.

Section 2: The Four Fundamentals

Clear Primary Objectives

What are we trying to prove or disprove? Are we demonstrating that the system can handle peak load? Finding bottlenecks through ramp testing? Assessing scalability limits?

Each test type serves a specific risk mitigation purpose. A spike test answers a different question from a soak test. An endurance test answers a different question from a capacity test. If you cannot state in one sentence what risk you are mitigating, you are not ready to test.

I once reviewed a project where the team had spent three weeks building a sophisticated load test. Beautiful scripts. Realistic data. Proper correlation. When I asked what they were trying to prove, the answer was "we need to load test before release." That is not an objective. That is a checkbox. They had no idea what response time was acceptable, what throughput was expected, or what failure mode they were worried about. Three weeks of work produced numbers that nobody could interpret because nobody had defined what "good" looked like.

Solid Planning and Preparation

No amount of AI can compensate for poor upfront thinking. You still need realistic load profiles, meaningful test environments, and representative test data.

Load profiles are where most teams cut corners. Real production traffic is messy: peaks and valleys, seasonal patterns, power users who behave nothing like the average. A flat load profile at 100 concurrent users tells you almost nothing about how the system will behave when 500 users hit the checkout simultaneously during a flash sale.

I worked on a retail project where the team tested with a steady-state load of 200 users for 30 minutes and declared the system production-ready. On launch day, 1,200 users arrived in the first 90 seconds. The database connection pool was sized for 200. The system queued every request after the pool filled, response times climbed to 45 seconds, and the load balancer started returning 502s. A spike test with realistic arrival patterns would have caught this in the first week of testing.

Test environments are the other common failure. If your test environment has half the database servers, a quarter of the data, and none of the CDN configuration of production, your results are fiction. You are testing a different system. You need to understand and document the deltas between test and production environments, and factor them into analysis.

Test data matters as much as test infrastructure. A search function that runs in 50 milliseconds against a thousand records may take 8 seconds against ten million. A report generator that performs well with a month of data may time out with a year of history. If your test data does not match production volumes, your results are misleading in a way that no amount of analysis can correct.

Proper Execution

Even with automated pipelines, someone needs to understand what the test covers and whether it makes sense. I have seen teams run a thousand virtual users against a system where the realistic peak was fifty. The test passed. The numbers looked impressive. The system fell over in production because nobody had asked whether the test scenario matched

reality.

Execution is not pressing a button. It is understanding the relationship between the script, the load model, and the business scenario. AI can automate the mechanics. It cannot replace the judgment of knowing whether the test you are running will answer the question you are asking.

Thorough Analysis

Pretty dashboards do not analyse themselves. Someone still needs to interpret results and translate them into business risk language.

A 95th percentile response time of 2.3 seconds is meaningless without context. Is that acceptable for an internal reporting tool? Probably. Is it acceptable for a customer-facing checkout page where every second of delay costs revenue? Probably not. Is it a regression from last month's 1.8 seconds? That matters. Is it caused by a database query that scales linearly with data volume and will be 4.6 seconds by next quarter? That matters more.

Analysis requires domain knowledge, system understanding, and the ability to connect a number on a graph to a business decision. AI can identify patterns in results data. It cannot tell your CTO what those patterns mean for the product launch.

The best performance analysts I have worked with share one skill: they can translate technical findings into business language. Not "the database connection pool is saturating at 300 concurrent sessions" but "we will lose approximately 15 percent of checkout transactions during the Christmas peak unless we increase the database tier by the end of November." The first statement is accurate. The second drives action. That translation is a human skill.

Section 3: The Usual Suspects

After twenty-five years in this field, I keep seeing the same failure patterns repeat, regardless of how modern the tech stack.

Vague or missing requirements. "It needs to be fast" is not a performance requirement. Without specific, measurable acceptance criteria tied to business outcomes, performance testing becomes an exercise in generating numbers that nobody acts on.

Unrealistic test environments or data. Teams do not understand the deltas with production, or worse, they understand them but dismiss them as "close enough." Close enough breaks under load.

Performance as an afterthought. Bolting it on at the end instead of building it into the architecture from the start. By the time a load test reveals an architectural bottleneck, the cost of fixing it is ten times what it would have been at design time. I have seen teams discover three weeks before launch that their synchronous payment processing pipeline cannot handle more than 50 transactions per second. Redesigning it for asynchronous processing required changes to six services, two database schemas, and the frontend notification system. The launch slipped by two months. A 30-minute architecture review at design time would have caught it.

Poor monitoring strategy. Flying blind without proper observability. If you cannot see what is happening inside the system during a load test, you cannot diagnose why it is slow. You can only observe that it is slow. And "it is slow" is not a diagnosis. It is a symptom. Without application-level metrics, database query times, queue depths, and thread pool utilisation visible during the test, you are guessing at the cause. And guessing at the cause means guessing at the fix.

No budget or time allocation. Treating performance testing as "nice to have." This is the failure pattern that enables all the others. Teams with no time and no budget skip the fundamentals because they have to skip something. And the thing they skip is almost always the thing that would have prevented the production incident they end up firefighting instead.

Lack of process awareness. Teams not understanding what it takes to build and maintain performant solutions. Performance testing is a discipline with a body of knowledge. It has principles, patterns, and proven practices developed over decades. When teams approach it as "run JMeter and see what happens," they get results that are not worth the electricity it took to generate them.

These are process and planning issues that no amount of automation can fix. I emphasise this because the rest of this book is about AI automation, and I do not want anyone reading it to think that buying a tool, any tool, is a substitute for thinking clearly about what you are testing and why.

AI can make your testing faster and more efficient. It cannot define your requirements or interpret your results. Those remain human responsibilities. The teams that get the most value from AI-powered testing are the ones who already have the fundamentals right. AI accelerates teams that already have good practice. It cannot create that practice for them.

Section 4: From Testing to Engineering

There is a broader shift happening in the industry that provides context for everything in this book: the move from performance testing to performance engineering.

Performance testing is an activity. You write scripts and produce reports. It happens late in the development cycle, after features are built and before release. The goal is verification: does the system meet its requirements under expected conditions? The output is a set of metrics, response times, throughput, error rates, measured against acceptance criteria.

Performance testing is essential. But it is reactive by nature. It validates what has already been built. If an architectural bottleneck surfaces during a load test, the cost of fixing it is high because the code is already written, integrated, and often close to production.

Performance engineering is a discipline. It starts at architecture and design, continues through development and CI/CD, and extends into production monitoring and capacity planning. It asks a different question: how do we make sure this system will perform, and keep performing, as it grows and changes?

Performance engineers do not stop at running tests. They shape system design, instrument code for observability, and feed production data back into development. The discipline requires cross-functional ownership, not a siloed QA gate at the end of the pipeline.

The traditional model, build everything then test at the end, worked when release cycles were measured in months. In modern delivery, teams ship daily or weekly. There is no time for a two-week performance test phase that blocks the pipeline. Late-cycle testing is expensive because defects found late cost more to fix. It is also incomplete because a single load test cannot replicate production traffic patterns and failure modes.

Performance engineering addresses this by shifting when and how performance work happens. Architecture reviews include capacity analysis. CI pipelines include lightweight performance checks on every build. Production observability feeds data back into test design, making capacity planning a continuous practice rather than a crisis response.

The industry has understood the shift from testing to engineering for years. The economics kept teams stuck in testing mode: scripting, correlation, and maintenance consumed so much effort that nothing remained for engineering work. Teams knew they should be doing architecture analysis and capacity modelling. They spent their time writing regex extractors and fixing broken scripts instead.

That is where AI changes the equation.

Section 5: Where AI Changes the Economics

The manual work that kept teams stuck in testing mode, the work that consumed 80 percent of the effort and left 20 percent for engineering, is the work that AI can automate.

Script creation. Recording a user journey and converting it into a runnable load test involves mechanical translation: HTTP requests, parameters, assertions. A browser recording produces a HAR file containing every network request. Converting that into a working JMeter or Locust test plan is a translation exercise: structuring requests into logical steps, adding think times, and generating assertions. AI handles this in seconds. A human takes hours.

Correlation. Identifying dynamic values, tracing their origins, and generating extractors to substitute them into subsequent requests. This is the single biggest time sink in performance test preparation. A complex enterprise application might have 50 or more dynamic values that need correlating. Each one requires detective work: find where the value originated, write the extraction rule, and verify the fix works. AI collapses this from hours or days into minutes. Chapter 2 covers this problem and its five levels of solution.

Script maintenance. When applications change, test scripts break. Dynamic values move and authentication flows evolve. In traditional performance testing, this creates a recurring maintenance tax. Teams invest weeks building a script portfolio, then watch it decay as the application changes underneath. Half the scripts break within a few sprints. By the end of the quarter, the portfolio is abandoned. AI-powered self-healing dramatically reduces this tax by detecting what changed and adapting the scripts. Chapter 8 covers how this works in practice.

Failure detection. Traditional pass/fail metrics miss a class of problems that only become visible when you look inside the responses. A login form appearing on a page that should show a dashboard. A JSON payload containing `"success": false` wrapped in an HTTP 200. An authentication redirect that returns the login page instead of the requested resource. AI catches these "soft failures" that human reviewers often miss.

Failure diagnosis. When a test fails, understanding why it failed is often harder than fixing it. AI can trace a 403 error back to a specific correlation failure, identify which dynamic value was stale, and show the engineer where the extraction should have happened. That diagnostic capability turns a 30-minute investigation into a 30-second read.

All of these are mechanical tasks requiring thoroughness and speed. Machines do them better than humans. Automating them frees teams to do the work that machines cannot: defining objectives, designing load profiles, and making engineering decisions.

That is the proposition of this book: AI replaces the mechanical parts of performance testing so that humans can focus on the engineering parts. The fundamentals in this chapter are the engineering parts. They do not get automated away. They become more important, because when machines handle scripting and correlation, the value you bring is in load profiles, analysis, and architecture decisions.

The teams I have worked with who adopted AI-powered testing saw a consistent pattern. Scripting, correlation, and maintenance time all dropped. But the teams that gained the most were the ones who reinvested that saved time into the fundamentals: better load profiles, more thorough analysis, earlier involvement in architecture discussions. The tools did not make them better testers. They freed-up time making them better engineers by enhancing their capabilities to mitigate more risk, more frequently.

Section 6: What This Book Covers

The rest of this book walks through how AI is changing performance testing in practice. Not in theory. Not as a marketing pitch. As a set of architectural decisions, engineering tradeoffs, and hard-won lessons from building a platform that does this for a living.

A note on scope: this book focuses primarily on HTTP and API-level performance testing, where correlation is the dominant challenge. Many of the principles apply more broadly, but the examples, tools, and workflows are grounded in HTTP traffic analysis and JMeter/Locust test plan preparation.

Chapter 2 tackles the oldest and most painful problem first: correlation. The Correlation Spectrum framework maps five levels of capability, from manual extraction through to specialised AI with persistent knowledge. Chapter 3 explains why correlation is not one problem but three, and why separating observation, judgement, and validation is the architectural insight that makes everything else work.

Chapters 4 through 6 are the practical core: the HAR-to-test workflow, the AI agents that power it, and the story of what happened when we gave one of those agents too much autonomy. Chapter 7 presents the evidence: time-motion data showing the real-world impact. Chapter 8 covers self-healing: what happens when tests break and AI fixes them.

Chapters 9 through 11 zoom out: the competitive field, how to build your own AI testing pipeline, and where this is heading next. Chapter 12 gets you started with your first AI-powered test.

But none of it matters if you skip this chapter. Clear objectives, solid planning, proper execution, thorough analysis. These are the foundations. AI amplifies good fundamentals. It also amplifies bad ones.

Get the basics right first. The machines will handle the scripting and correlation.

Chapter 2: The Correlation Problem, Performance Testing's Oldest Enemy

Section 1: The Problem That Will Not Go Away

Chapter 1 established a principle: AI cannot fix bad fundamentals. Clear objectives, solid load profiles, realistic environments, meaningful analysis. Get those wrong and no amount of automation saves you.

But there is a fifth fundamental that deserves its own chapter, because it is the single biggest reason performance testing projects stall or get abandoned. It is called correlation. And it has been the performance tester's oldest enemy for over twenty years.

Every performance tester knows the feeling. You record a user journey, hit replay, and watch your script crash within seconds. The culprit is almost always the same: dynamic data. Session tokens, CSRF values, and authentication keys all change between requests. If your script replays the values it recorded rather than extracting fresh ones from server responses, it is dead on arrival.

The process of fixing this, identifying dynamic values, finding their origin in a previous server response, and extracting them for reuse, is correlation. It is, without exaggeration, the single most time-consuming and frustrating part of performance test preparation. Industry surveys rank it as the number one pain point. A simple script might need a handful of correlations. A complex enterprise application, think Salesforce, SAP, or a modern microservices checkout flow, can require dozens or even hundreds.

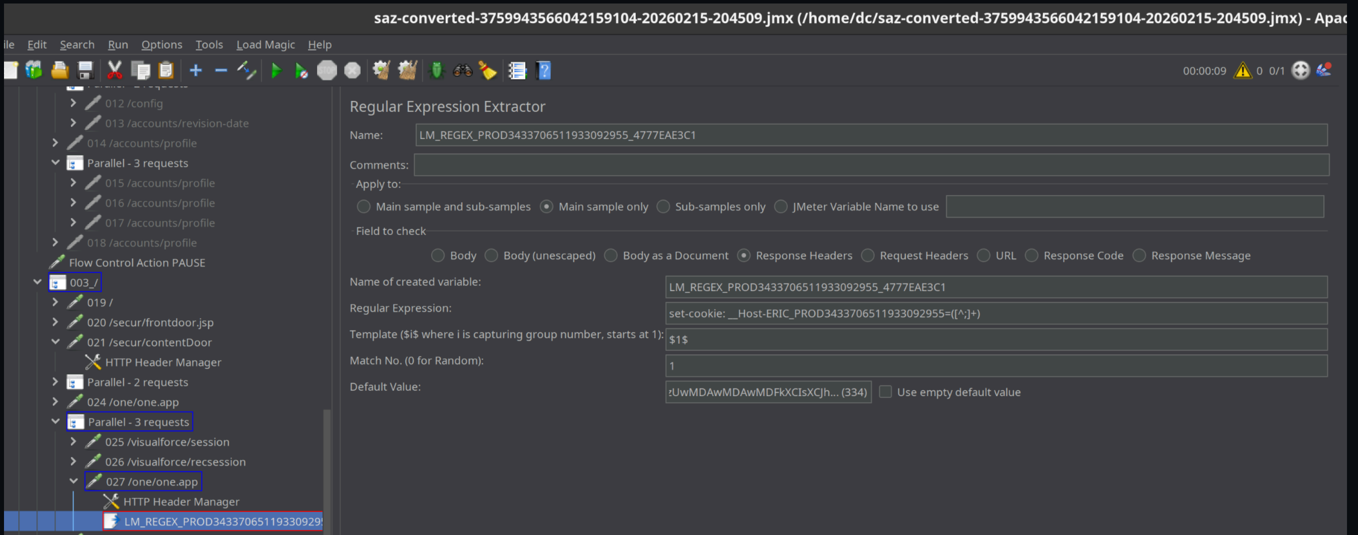
I have spent the better part of my career dealing with this problem. First as a tester wrestling with it, then as a consultant helping teams dig out of correlation backlogs, and now as the founder of a platform built to solve it. Along the way, I developed a framework for thinking about the different approaches the industry has taken. I call it the Correlation Spectrum: five levels of capability, from fully manual through to fully autonomous.

Understanding these levels is not an academic exercise. Choosing the right approach for your context determines whether your performance testing programme is viable, efficient, or too expensive to maintain.

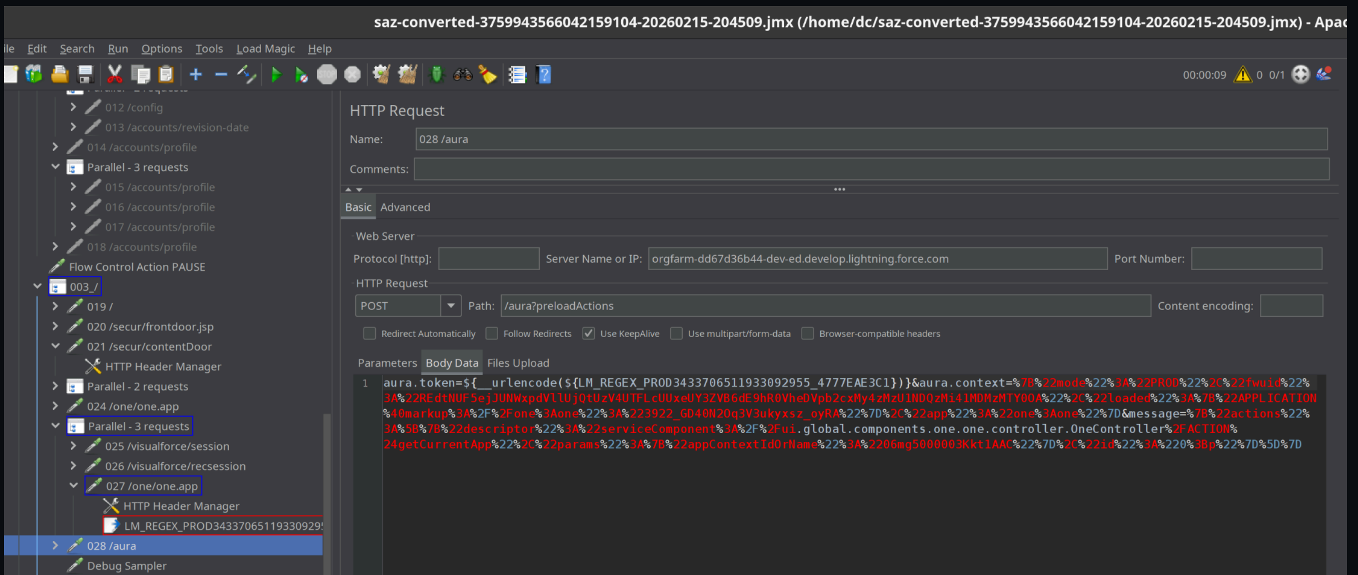
Section 2: Level 1, AI-Assisted Manual Correlation

An engineer records a user journey, replays it, and identifies which values need correlating. The tooling provides some intelligence in the background: auto-complete suggestions, highlighting of likely dynamic values, basic pattern recognition. But the human drives every decision.

In practice, the engineer opens the recorded script, often containing hundreds of HTTP requests. They look at a failing request, compare the recorded response to the replayed response, spot a value that has changed, then manually search backward through prior responses to find where that value first appeared. They write an extraction rule (a regular expression or boundary-based extractor), insert it after the originating response, and replace the hardcoded value in the failing request with a variable reference.



Modern tools at this level may use lightweight AI to suggest "this looks dynamic" or to auto-generate a regex pattern once the engineer has identified the target. But the core workflow, the detective work of tracing where a value came from and deciding what to do about it, remains human-driven.



This approach works for simple applications with a small number of dynamic values: five to ten correlations per script. APIs with straightforward token refresh flows, internal tools with basic session management, or applications where the tester knows the architecture inside out. The manual approach also gives the engineer maximum control and visibility, which matters in regulated environments where every test artefact must be explainable.

The problem is scale. Manual correlation effort does not grow linearly with complexity. It grows exponentially. Each additional dynamic value increases the search space and the likelihood of cascading errors, where fixing one correlation breaks another. Research suggests that for scripts with thirty or more correlation candidates, manual effort can consume forty or more hours per script. At that point, teams often abandon scripts rather

than maintain them. I call this the "Script Museum": test assets that sit unused because they are too expensive to keep current.

Typical time investment: minutes per correlation for simple cases. Hours per script for moderate complexity. Days to weeks for enterprise applications.

Best suited for: Small teams, simple applications, regulated environments where auditability is paramount, or situations where engineers know the application under test inside out.

Section 3: Level 2, Rules-Based Correlation Frameworks

The tool ships with a library of predefined correlation rules, organised by web framework or technology. When you record against a .NET application, for example, the tool recognises known dynamic parameters like `__VIEWSTATE`, `__EVENTVALIDATION`, and ASP.NET session IDs, and applies preconfigured extraction rules.

During or after recording, the tool scans requests and responses against its rule library. Each rule is a template: "For applications using framework X, look for parameter Y in response Z, and extract using pattern P." Leading commercial tools, including LoadRunner, BlazeMeter, NeoLoad, and OctoPerf, all implement some version of this. Some publish the specific web frameworks they support, giving you a compatibility list. If your application's technology stack is on the list, you get a significant head start.

Rules-based correlation works well for well-known technology stacks with predictable dynamic data patterns. A standard Java Spring application with CSRF tokens, a .NET application with ViewState, or an OAuth 2.0 flow using standard grant types. Industry reports suggest success rates of 60 to 90 percent for applications that match the rule library.

The limitation is inherent in the model: rules only work for patterns the vendor has already seen and codified. Custom frameworks, bespoke authentication mechanisms, or any technology not in the rule library will be missed. The tester then falls back to manual correlation for everything the rules did not catch, and diagnosing why a rule did not fire (or fired wrong) can be more frustrating than doing it by hand in the first place.

There is also a maintenance burden that is easy to overlook. Framework updates and evolving security patterns mean the rule library needs constant updating. If your vendor falls behind, or if you are an early adopter of a new framework, you are on your own.

Expect to handle 60 to 90 percent on well-matched stacks, then spend manual effort on the remainder. That last 10 to 40 percent often represents the hardest, most time-consuming correlations.

Best suited for: Teams working with mainstream commercial platforms, organisations with predictable technology stacks, and environments where consistency across testers matters more than handling novel scenarios.

Section 4: Level 3, Code-Assisted Correlation Automation

The tool uses algorithmic analysis: diffing engines, heuristic pattern matching, and statistical anomaly detection to identify dynamic values without relying on a predefined rule library. It compares multiple recordings or replays, spots values that change between executions, and generates extraction logic.

Rather than matching against known frameworks, this approach compares a request parameter's value across two or more recordings of the same journey. If a value is identical across recordings, it is likely static. If it changes, it is a correlation candidate. The system then searches prior responses for the changed value's origin, applies smart algorithms to determine the best extraction method (regex, JSON path, boundary match), and inserts the correlation.

More sophisticated implementations add statistical scoring to rank candidates by likelihood, use response structure analysis to narrow the search space, and apply encoding detection to handle Base64, URL encoding, or nested values. Some tools at this level are incorporating machine learning classifiers trained on historical correlation data.

This is the first level that is framework-agnostic. Because it does not rely on predefined rules, it can handle custom applications and bespoke frameworks. The algorithmic approach also scales better than manual work: doubling the number of correlation candidates does not double the analysis time the same way it does for a human.

But the algorithms are smart, not intelligent. They can tell you what changed, but they struggle with why it changed and whether it matters. False positives, flagging static values that happen to differ between recordings due to test data differences, require human review. False negatives, missing dynamic values that happen to look similar across recordings, cause failures that are difficult to diagnose.

Complex scenarios also challenge this approach. Values assembled client-side from multiple server responses, tokens that appear in encoded form, and cascading correlation chains where extracting value A is a prerequisite for extracting value B. All of these require contextual understanding that pure algorithmic analysis lacks.

And there is no learning between sessions. Each new script starts from scratch. The tool rediscovers patterns it has seen before and makes the same mistakes it made last time.

Expect significant acceleration on first-pass correlation, but plan for a review and correction phase that can consume 20 to 40 percent of the total effort on complex applications.

Best suited for: Teams with diverse application portfolios, API-heavy architectures, and organisations that have outgrown rules-based tools.

Section 5: Level 4, AI-Driven Correlation (General-Purpose AI)

Large language models and general-purpose AI enter the correlation workflow. Rather than relying on algorithmic pattern matching, the tool uses AI to understand the semantic meaning of requests and responses, reason about why a value might be dynamic, and generate extraction logic with contextual awareness.

The AI receives the recorded traffic, or a structured representation of it, and analyses it with a level of comprehension that goes beyond pattern matching. It can read a JSON response and understand that `"csrfToken": "abc123"` is a security token that will change per session. It can trace authentication flows and determine that the `Authorization` header in request 47 should contain the bearer token returned in response 12, or examine error messages from failed replays and reason about the probable cause.

Implementation involves integrating an LLM (such as GPT, Claude, or Gemini) into the tool's workflow, either through API calls or embedded models. The AI may generate regex patterns, suggest JSONPath expressions, or modify the script. Some implementations create a conversational interface where the engineer can ask the AI to "correlate the session token" or "fix the authentication flow."

General-purpose AI brings genuine comprehension to the problem. It understands that an OAuth `access_token` and a custom `x-auth-key` serve similar functions, even though they look different. It can handle novel application architectures without predefined rules because it reasons from first principles rather than pattern libraries. The conversational interface also lowers the barrier to entry. Junior engineers who struggle with regex syntax can describe what they need in plain language and get working extraction logic back.

But general-purpose AI was not built for this problem, and it shows.

First, context window limitations. A complex recording might contain thousands of requests and responses, each with headers, cookies, and body content. Feeding all of this into a general-purpose LLM either exceeds context limits or forces aggressive summarisation that loses critical detail. The AI might identify that a token needs correlating but miss the specific response where it originated because that response was truncated.

Second, no accumulated knowledge. Each session starts fresh. The AI does not remember that last week it correlated the same Salesforce `aura_token` pattern, or that a particular CDN's response headers always contain a specific dynamic value.

Third, general-purpose AI lacks the specialised understanding of performance test tooling. Generating a regex is one thing; generating a regex that is safe for JMeter's regex extractor, which has specific syntax requirements and boundary handling quirks, is another. An AI that suggests a valid regular expression which happens to use a lookahead unsupported by the target tool creates more debugging, not less.

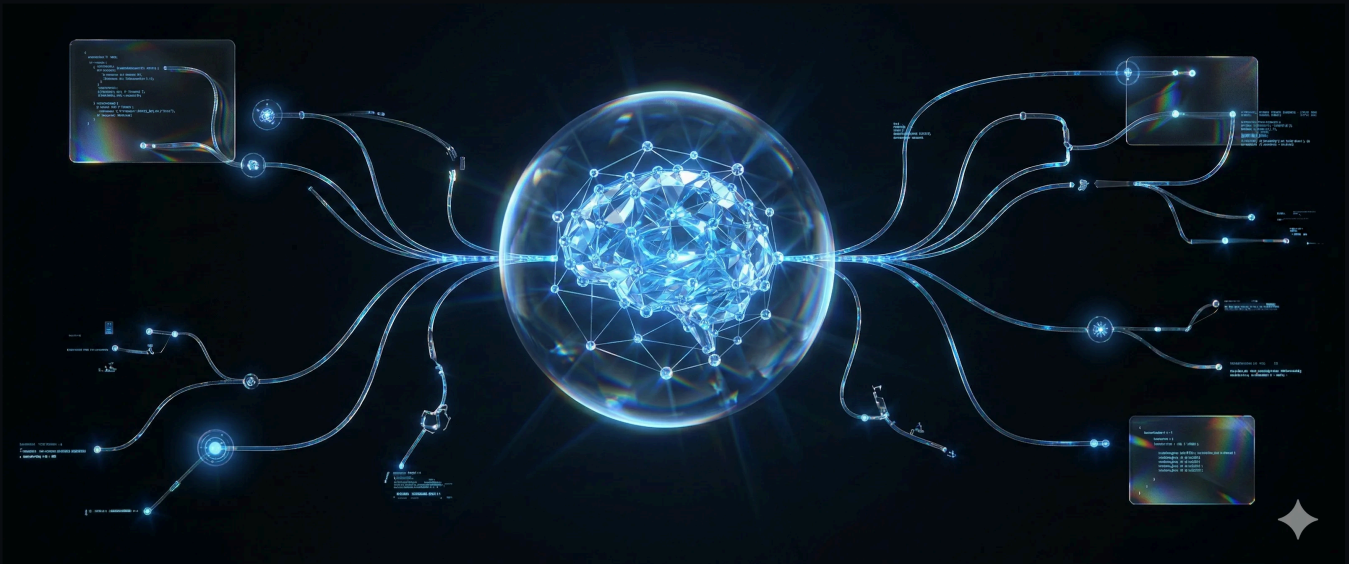
Fourth, hallucination risk. LLMs sometimes generate plausible-looking but incorrect extraction logic: a JSONPath that references a non-existent field, or a regex that matches the right value in the example but also matches three other values in production. Without domain-specific validation, these errors propagate without warning.

The result is fast initial output but variable quality. Expect to spend significant time validating AI output on complex applications. The "review and fix" phase can sometimes approach the time manual correlation would have taken, more so when the AI's errors are subtle.

Best suited for: Teams already using AI tooling, one-off correlation tasks on unfamiliar applications, and environments where the speed of a "good enough" first pass justifies the validation overhead.

Section 6: Level 5, Specialised AI with Persistent Knowledge

This is what we are aiming for with LoadMagic. And this is what I designed, because every level described above left me frustrated in a different way.



Level 5 is a purpose-built system where specialised AI agents, enriched by a persistent knowledge base of correlation patterns, work in concert with smart code to handle correlation end to end. Detecting dynamic values, extracting and substituting them, validating the results, and learning from every session.

The system operates on a different model. Rather than treating each correlation task in isolation, it builds and maintains a living knowledge base: a persistent, structured memory of every correlation pattern it has encountered, organised by application, technology, and confidence level.

When a new recording is imported, the system first consults this knowledge base. If it recognises the application (by domain, API structure, or technology fingerprint), it pre-applies high-confidence extractors before the first test even runs. The system pre-excludes known noise domains, flags encoding requirements, and highlights tricky patterns like client-assembled values or multi-step token refresh chains. Much of the correlation work is done before a single error is encountered.

For values that are not in the knowledge base, the system shifts to error-driven detection. Specialised AI agents analyse failures in context. Not just "this request returned a 403" but "this request returned a 403 because the CSRF token in the request body does not match the one the server expects, and the server's expected value was returned in the response to request 23, in a JSON field at `$.meta.csrfToken`." The agents understand the full chain.

The agents themselves are specialised rather than general-purpose. A correlation agent identifies candidates and determines the extraction strategy. If the target value lives in a JSON response, it applies a JSONPath extractor. If a more complex boundary match is needed, it delegates to a regex specialist agent that builds expressions with proper boundaries and tool-safe syntax. A QA agent validates the result against best-practice standards. Each agent does what it is best at. You will meet all of them in Chapter 5.

After successful correlation, the results feed back into the knowledge base with confidence scoring. Patterns that work across multiple sessions gain confidence. Patterns that fail are downgraded or replaced. When applications change, a platform update that alters token locations or authentication flows, the system detects the structural drift through schema fingerprinting and adapts, rather than breaking without notice.

This is the self-healing dimension. If a working extractor breaks during a subsequent test run, the system does not just flag the error. It investigates. It identifies whether the failure is due to a changed response structure, a relocated value, or a new dynamic parameter. It applies a fix. The engineer sees the fix, not the failure.

For organisations where performance testing is a recurring, ongoing activity rather than a one-off event, the compounding effect changes the economics. The system gets faster and more accurate with every session. Time-motion analysis has shown reductions from weeks of manual effort to hours, with the gap widening as application complexity increases. Chapter 7 presents the detailed numbers.

Best suited for: Enterprise teams with large script portfolios, organisations testing complex commercial platforms, teams where performance testing is a continuous practice.

Section 7: Comparing the Five Levels

Dimension	Level 1: Manual	Level 2: Rules-Based	Level 3: Code-Assisted	Level 4: General AI	Level 5: Specialised AI
Detection method	Human-driven with AI hints	Template matching	Algorithmic diffing	LLM semantic analysis	Knowledge-first with error-driven fallback
Framework dependency	None (limited by engineer knowledge)	High (vendor rule library)	Low (framework-agnostic)	None (limited by context window)	None (framework-agnostic + accumulated knowledge)
Learning between sessions	Human memory only	None	None	None	Continuous (persistent knowledge base)
Self-healing	None	None	None	Limited	Fully automated

Dimension	Level 1: Manual	Level 2: Rules-Based	Level 3: Code-Assisted	Level 4: General AI	Level 5: Specialised AI
Scalability	Poor (exponential effort)	Moderate	Good (linear scaling)	Variable (context window constrained)	Excellent (compounds over time)
Typical success rate	100% (eventually)	60-90% on matched stacks	70-85% first pass	60-80% (before validation)	90%+ (improving over time)
Cost profile	High recurring labour	Moderate licensing + gap labour	Moderate licensing + validation	API costs + validation labour	Platform investment, low recurring effort

Section 8: Choosing the Right Level

There is no single best approach. Only the best approach for your situation.

If you are a small team testing a handful of simple applications with stable architectures, Level 1 or Level 2 may be sufficient. The investment in more sophisticated tooling does not pay back when you are correlating five values per script once a quarter.

If you work across a diverse set of technologies and need framework-agnostic detection but are comfortable with a review-and-adjust workflow, Level 3 gives you meaningful automation without platform lock-in.

If you are experimenting with AI across your engineering practice and want to extend that to performance testing, Level 4 can deliver quick wins. Go in with eyes open about the validation overhead and the absence of accumulated learning.

If performance testing is a core, ongoing practice for your organisation, if you maintain dozens or hundreds of scripts and your applications change often, the compounding intelligence of Level 5 is where the economics shift. The upfront investment in a specialised platform pays back fast, and the gap between it and every other approach widens with every session.

Manual correlation becomes less tenable as applications grow more complex. Rules-based approaches hit a ceiling that no amount of rule-writing can raise. The systems that will win combine domain expertise with persistent, compounding intelligence: they solve the correlation problem, remember the solution, and get better at it over time.

But knowing the five levels is not enough. Level 5 works because of a specific architectural insight: correlation is not one problem. It is three. The next chapter explains why that distinction matters, and why most tools get it wrong by treating correlation as a single monolithic task.

Keep Reading

Thanks for reading this sample of *AI Performance Engineering*.

The full book covers ten more chapters, including:

- The five AI agents - George, Carrie, Rupert, Suzy, and Quinn - and how they work together (Chapter 5)
- The God Mode story: what happened when we gave one agent too much autonomy, and what we learned (Chapter 6)
- A stopwatch time-and-motion study - manual correlation versus LoadMagic, with minute-by-minute data and projections out to enterprise scale (Chapter 7)
- Self-healing tests and the quality gate (Chapter 8)
- A map of the AI testing landscape and the four approaches you can choose between (Chapter 9)
- A blueprint for building your own AI testing pipeline (Chapter 10)
- Where AI testing is heading in the next two to three years (Chapter 11)
- A practical getting-started guide (Chapter 12)

Plus three appendices: FAQ, comparison table, and glossary.

Where to go next

- **Full book on Leanpub:** leanpub.com/ai-performance-testing
- **Try LoadMagic:** loadmagic.ai
- **Connect with me:** linkedin.com/in/davidcampbell-ai

If this sample sparked a question, a disagreement, or a "we do it differently" story - I would love to hear from you.