SACRA

EXPERT INTERVIEW

UPDATED
06/04/2025

Wei-Wei Wu, CEO of Momentic, on AI-native end-to-end testing

TEAM

Jan-Erik Asplund
Co-Founder
jan@sacra.com

Published on **Jun 04th, 2025**

Wei-Wei Wu, CEO of Momentic, on AI-native end-to-end testing

By **Jan-Erik Asplund**

Background

We reached out to Wei-Wei Wu, co-founder of Momentic ($3.7M seed, FundersClub), an AI testing tool, about how “vibe testing” is replacing brittle test suites with continuous, AI-generated code validation.

Key points from our conversation via Sacra AI:

- **In the early 2000s, end-to-end software testing required a QA team manually clicking through an app to catch bugs—Selenium (2004), Cypress (2017), and Playwright (2020) emerged to enable developers to write scripts that live in their codebase and programmatically simulate user actions like clicking, typing and submitting forms, but because they rely on hard-coded IDs, CSS, and XPaths, even minor UI changes cause test suites to fail, leading teams to constantly rewrite tests or ignore them entirely**. “Playwright, Cypress, and Selenium are very tightly coupled to how your app is built... Tests become flaky because the moment your application changes—front-end teams are making updates and shipping fast—it's very common to forget to update your tests. Now your tests have some hard-coded attributes that no longer map to an element on the page."

- **Vibe coding is accelerating the output of both code and bugs, creating the need to get test coverage across new features faster and driving the rise of developer-native "vibe testing" tools like Momentic, Antithesis ($77M raised, Amplify Partners), QA Wolf ($56M raised, Inspired Capital) and Qodo ($50M raised, Susa Ventures) that translate natural language into code and bundle in test generation, auto-repair when the UI changes, and observability**. "With a lot of vibe coding using tools like Cursor or Windsurf, you're telling the AI in the editor what you want to build in natural language. We essentially let you turn those instructions into a repeatable automation that you can run on your CI as a blocking check to make sure the feature you're building or updating stays up-to-date and is always working."
- **As writing end-to-end tests becomes faster and easier, it's moving out of QA teams and into the entire engineering organization, turning testing from post-hoc validation to always-on developer infrastructure and creating the opportunity to merge together once-separate categories like CI test checks (GitLab), browser automation (BrowserStack), observability (Datadog) and QA outsourcing (Applause) into one unified platform for validating that software actually works**. "There have been studies on who owns QA and how that improves your metrics. If you want a high-efficiency engineering team, you want engineering to own testing. There's no substitute. Historically, QA has always been a separate organization because teams think engineers don't have enough time to do QA… But in the past few years, we're seeing a big shift with high-caliber engineering teams. They don't have separate QA organizations—engineering owns QA."

Interview

In short, what is Momentic?

Momentic is a year and a half old. We help engineering teams test their web products—things like web apps or anything people can load in a browser. I used to do a lot of developer tooling and open source work. If anyone has ever used

Node.js, you've run my code since version six, which has been a couple years ago.
One of the biggest pain points with every engineering team I've been on has always been the end-to-end tests. These typically use tools like Selenium, Cypress, or Playwright. They take a lot of engineering time to build. More importantly, as your test suite grows and your product evolves, your test suite becomes quite unmaintainable.

We've heard stories of teams where they have flaky tests they'll just rerun until they pass. That wastes a lot of time and isn't great for engineering productivity. It slows down your product velocity and creates a poor experience for everyone involved. We started Momentic to help make this process automated, as easy as possible, and of course, as reliable as possible, so the signals you get out of our tests are high quality and engineers can trust the results.

Can you speak a little bit to what makes testing for the web complicated? Why do tests become brittle and flaky over time?

Test automation tools like Playwright, Cypress, and Selenium are very tightly coupled to how your app is built. One of the core components of any test automation tool is it must be able to interact with the web app. For instance, I want to click a button, interact with a modal, or type a value into an input field.

How you target those elements in these tools is very tightly coupled. You're using hard-coded IDs, CSS selectors, or XPaths to target elements on the page. Tests become flaky because the moment your application changes—front-end teams are making updates and shipping fast—it's very common to forget to update your tests. Now your tests have some hard-coded attributes that no longer map to an element on the page.

That causes flaky tests. You'll have a test failure that's not actually an issue with your application but an issue with your test script. This decreases trust and increases maintenance overhead.

Momentic lets developers write end-to-end tests in plain English and skip all the brittle selectors—what was the breakthrough that made that kind of UX actually reliable?

Instead of using hard-coded attributes, one of the core unique parts about Momentic is you can use natural language to target and describe what you want to do on the web app. We're not tied to how your frontend code is implemented. We don't care what component library you're using or what framework—React, Vue, Angular, whatever you want.

We let you capture user intent on how you want your users to use your application, and we help you turn that into a repeatable automation that can run in your CI pipelines, on a schedule, anywhere you want to run these tests.

Is vibe coding a tailwind for Momentic?

With a lot of vibe coding using tools like Cursor or Windsurf, you're telling the AI in the editor what you want to build in natural language. We essentially let you turn those instructions into a repeatable automation that you can run on your CI as a blocking check to make sure the feature you're building or updating stays up-to-date and is always working.

We help bridge the gap where with AI coding tools, the rate of change is increasing significantly. We're seeing organizations able to ship really fast. But at scale, shipping fast isn't the only thing that matters. When you have customers and SLAs, quality is extremely important—a first-class citizen that's top of mind for every engineering leader. We bridge the gap for engineering teams that are vibe coding but still need robust testing, because quality is the only thing that matters.

Updates are part of the problem. Whenever you're shipping so much code, another big problem is that you don't have coverage on all these new features the teams are shipping. We help them solve both problems. Your existing tests can be updated automatically, and for new tests, we can help you generate them based off of your natural language instructions.

It's very low effort for an engineer to actually use Momentic to get the quality you want. Otherwise, you spend a couple hours writing a Playwright test, check it into your code base, and then a couple days later, someone breaks it, and now you have a CI pipeline that's flaky.

Where in the development cycle do Momentic tests usually run—blocking the merge in CI, gating production deploys,

or continuously in prod—and why?
People have different approaches to their QA strategy, but generally, the best engineering teams we work with are shifting left. That means having tests as early in the coding process as possible.

This comes into play because you can run Momentic tests locally. They're checked into GitHub. You're running these as blocking checks to make sure critical flows aren't broken when you merge into main. That's the ideal situation—running Momentic in every part of your developer workflow to make sure things aren't broken, because that's ultimately what matters.

Inside a typical customer, who actually writes and maintains Momentic tests—frontend devs, a central QA team, or even product managers?

There have been studies on who owns QA and how that improves your DORA metrics—DevOps metrics like mean time to repair or mean time to recovery. If you want a high-efficiency engineering team, you want engineering to own testing. There's no substitute.

You don't want QA, PMs, or designers to own testing. You want engineers who are building the code—they have GitHub access, they're creating pull requests—you want them to test. Rather than a separate QA organization, which you'll see in many large older organizations.

There's a great book called "Accelerate" that talks about how engineering team structure can impact your DevOps metrics. Your team will move fastest and have more ownership when the people writing the code and features also own the testing and quality. You have higher ownership and higher velocity because of shorter turnaround times. You don't have to toss it over the wall to QA. And ultimately, you have happier customers because your product isn't broken.

Historically, QA has always been a separate organization because teams think engineers don't have enough time to do QA. Many organizations spin up separate QA teams as a response. But in the past few years, we're seeing a big shift with high-caliber engineering teams. They don't have separate QA organizations—engineering owns QA.

Engineering leaders want this. As the founder of a startup, I'm not looking to hire QA engineers. Our engineers should own the quality of what they ship. That ownership is really important.

How does Momentic coexist with unit-test frameworks (Jest, Vitest, etc.)—do teams still need them?

That's a great question. Historically, we've had the testing pyramid. At the bottom, you have granular unit tests for individual functions in your back-end or front-end code. Since they're so granular, you have to write a ton of unit tests—they're easy to write but high volume. They form the bottom of the pyramid.

In the middle, you have integration tests and API tests—that's the balance between effort and value provided. At the very top of the pyramid, you've always had your end-to-end functional tests, using your app and hitting all the back-ends to make sure everything is working. For example, testing your Stripe payment flow—payments always need to work.

This has historically been the smallest part of the pyramid because it was very high effort, and the perceived value wasn't that high because tests were flaky. You'd get a lot of false positives, and with flaky tests, engineers just ignore the alerts after a while.

Now, since we're able to make this process really easy and reliable, I see this becoming a much larger part. I honestly think the pyramid can invert. End-to-end tests can cover most use cases because you're actually clicking through your app to make sure it's not broken. You can have 100% unit test coverage, but if your app isn't working, does that actually matter?

The end goal isn't how much code coverage you have—it's whether your customers have a good experience. The best way to ensure that is to simulate those user experiences through functional end-to-end tests.

What does pricing look like for Momentic—do you think the future model is seat-based, usage-based, or something more outcome-oriented?

Our current pricing model is usage-based, very similar to if you're using Anthropic or OpenAI with a credit-based system for test execution. That's how we power auto-healing for your tests, how we use AI to validate things on your page, or generate tests for you.

I could see a world where we focus on value-based testing. Sierra was pretty famous for this—they billed on the number of tickets resolved. For us, a potential avenue could be defects caught or bugs caught before merging to main or before deployment. But that's just something to explore.

What are all the open source or commercial tools that teams are using for testing right now that Momentic is either replacing or augmenting in the stack?

We primarily work with engineering teams, and the most common tools we see them use are typically open source. This would be Selenium, which turned 20 years old last year, Cypress, which is open source, and Playwright, which is open source and sponsored by Microsoft. Those are typically the tools we help teams replace.

One of our customers saw a 5x speed improvement in time to automate moving from Playwright to Momentic. We were able to see some very drastic reliability improvements moving from Cypress to Momentic as well. One customer reported a flaky test that had around a 40% pass rate over a two-week period. After migrating that test to Momentic, that same test achieved a 99.35% pass rate on main. The only thing that changed was the test automation tool, not the application under test.

If I had to make a market map, the testing industry is quite large and quite old. You have a lot of outsourcing. You have incumbent tools like Tricentis and SmartBear—they've done many acquisitions in this space. They bought Reflect and Testim. There are many low-code, no-code testing tools that primarily target non-technical personas. They're typically selling to product or QA teams, sometimes even support teams.

One key thing with Momentic is we're helping the engineering teams and organizations we work with drive a shift of QA ownership onto engineering because it's so effortless. As a result, you're moving faster, shipping faster, shipping fewer

bugs, and everyone's happier. Your customers are happier because you're fixing their bugs quickly or they never encounter bugs due to comprehensive testing.
In the market, there are definitely many incumbent tools—a mix of software and outsourcing services.

What is the typical flow for the engineering team using Momentic, where they build a feature and then they have to build a test? What does that look like?

We work completely locally. We have a package on NPM called Momentic—that's our test runner and local app. When you're building a new feature, you'd start up the Momentic app, use our editor to record or build your test, and check that into GitHub. You can run it in your CI/CD pipelines. Momentic can run anywhere that supports Node 18, which is already a pretty old version.

That's the typical flow of how we integrate with our customers. You can run your tests locally to make sure your feature is working on your local dev server, run them in CI, and most commonly, these run as blocking checks on pull requests. Your set of critical tests must pass before you merge into main, so you don't risk merge regressions.

There are different ways you can build a Momentic test. You can import your existing scripts from Cypress or Playwright—we help you migrate those over automatically with our AI. You can click around on the browser window to automatically generate the test.

Or, if you're like me and prefer fine-grained control of what the test actually does, you can use our editor and natural language to describe exactly what you want the AI to do on your page. Tests are best when they're very tightly scoped and well defined, or else you'll run into nondeterministic tests. If you don't know what you're testing, how is the AI supposed to know?

Playwright and Cypress—what exactly did they do right, and how did they become these testing suites with such big adoption, and where are they lacking?

The fundamental primitives of these tools are a little different. With Cypress and Playwright, you target elements on the page

using hard-coded IDs, accessibility attributes, placeholders, ARIA labels, and similar approaches.
With Playwright and Cypress, which I consider newer tools, they do some things well—they have the concept of auto-waiting and actionability checks. They'll automatically wait to see if an element you want to interact with is actionable—can you click on it, can you type into it. Selenium didn't have this built in; you'd have to orchestrate it manually yourself.

Coming from Selenium, Playwright and Cypress offer a much better developer experience. They're easier to write and build, and they support various languages. I work primarily with full-stack TypeScript, so Playwright's TypeScript support is immaculate—it's perfect.

They've done a great job of integrating with developer workflows. You run Playwright locally, store it in your source control, and any engineer can edit it. They're also extremely powerful—you can use them not just for testing but for general workflow automation, RPA, or web scraping. If you want to get that Pokémon card on Nintendo's website when they launch, people use tools like Playwright to write bots.

Where it falls short is that it's tightly tied to your application implementation—your IDs, placeholders, and attributes that are hard-coded. It becomes very brittle because the moment your application changes, it's very common to get flaky tests. With actionability checks, they've alleviated some flakiness, but the core problem remains—they're still relying on these attributes to target elements.

Another issue, not necessarily tied to any specific automation tool, is that end-to-end tests span your whole stack—front-end, back-end, database, and whatever other microservices you're running. There's a lot of variability. For example, if you're logging into Facebook and a network request takes 30 seconds longer than expected, the test would fail because the predetermined wait time wasn't long enough.

There are external factors outside the automation tool's control that can impact test execution. Because it's end-to-end and covers such a large surface area, there are many things that can go wrong. Your database might go down, your back-end server might take too long to respond, you might get anti-bot blocked, or a new marketing dialog pops up unexpectedly and fails the test.

GitHub is weaving Copilot-generated tests into VS Code and Azure DevOps—what do you make of this attempt to bundle AI testing into the IDE layer?

You could expose an MCP server that Cursor could connect to. We have a CLI with commands you can use, similar to coding agents like Codex or Cloud Code. The core component is that you need to meet developers where they are, and it's rarely on a cloud-hosted app. It's locally in their IDE, on localhost, in their GitHub, in their pull requests.

Some teams are trying to fold E2E testing into observability and monitoring—do you think those stacks converge?

At Momentic, we have built-in scheduling and alerting mechanisms for monitoring use cases. These are called browser synthetics—usually critical flows that aren't very complicated but make sure your app is working. For example, being able to log in or log out, or just accessing the page.

The underlying tooling is honestly the same. We have many customers coming to us with existing synthetics tools. Datadog has a product I've used in the past, but historically, it's flaky. With any sort of testing, you don't want it to be flaky because that creates noise for your engineers. When it's notoriously flaky, it's like the boy who cried wolf—when it actually fails, people won't notice because they ignore the alerts.

How do you think about the role of QA engineers in a world where developers are increasingly responsible for writing and maintaining their own tests? Does something like Momentic eliminate the need for QA teams?

I don't think so. The barrier for coding is much lower now. You don't need to go to college and get a CS degree to function as an engineer—you can be a great engineer without a CS degree. More people are going to become engineers, and we're enabling those engineers to easily test, regardless of whether they used to be in QA, product, or support. The barrier for entry to engineering is much lower, and we're helping engineers build and ship reliable products.

So QA engineers will just become software engineers?

Exactly. For the AI development lifecycle to function autonomously, right now the loop goes: human instructs Cursor, Windsurf, or Devin to output code. There's a big gap of how to test this. Maybe you generate some unit tests, run static checks like linters or type checking, but the feedback loop is limited to what is generated. In some cases, there's access to a browser to view what it's building, like v0 generating front-end code.

We want to help AI engineers close that gap. Imagine you vibe-coded something with Devin and deployed it somewhere. You want to validate all the critical flows that might not have been intended to change. Every team maintains hundreds or thousands of critical flows that must never break. Engineers might think they only need to test the feature they just built, but breaks often happen elsewhere.

We provide that feedback loop into AI coding agents and editors so that whatever changes you make, you can be confident all required checks are passing and critical flows are working. That feedback loop goes back to your coding tool, which generates fixes if needed. If everything's green, you're good to go.

The goal is shifting so far left that your code doesn't even have to be committed—it could just be changes on your local file system, and we can help you test it because Momentic runs completely locally.

If Momentic succeeds, does the total addressable market shift from “QA engineers at SaaS companies” to “every developer who ships code”?

Yes, everyone is empowered to own the testing—anyone who makes code changes.

There's a separation between your user flow definition (what teams call test cases) and your automation. Momentic is the automation, but we also provide mechanisms for teams to document how user flows are supposed to function. The test cases are your source of truth.

This is usually a collaboration between teams—product, design, engineering, support. Everyone can contribute to this source of truth, but it needs to be separate from the automation implementation because your implementation might have bugs or be out of date.

That's where non-engineering teams can contribute to testing—they provide the source of truth for how things are supposed to work, and the automation is just implementation details. The test case might be "log in with username and password," while the automation breaks that down to typing an email, typing a password, and pressing login—different levels of abstraction.

Engineers who are part of product decisions and write the code create the automation but also contribute to the test case source of truth. A PM who wrote a PRD specifying user flows contributes to the test cases but isn't writing the automation—they just decide how things should work for users.

When someone generates a test in Momentic using natural language, does it stay natural language, or is it exposed back to the user as the actual code?

We don't generate any code. A Momentic test uses our own DSL, but it is in natural language—just lower level. For example, instead of saying "go to Instacart and buy the first thing on the suggested list," it breaks that down into granular steps because you want your test to be deterministic.

No engineer likes nondeterministic tests or pipelines. You want to know exactly what your test is doing in the browser. We break high-level instructions into granular steps—click, type, scroll, drag and drop. You store that test in your source control and run it deterministically in your CI/CD pipeline.

The test case serves as the high-level instruction. If the test case changes, our AI can automatically update the automation. It's important to have these be separate, because one serves as the source of truth, one as the implementation. The test case may not change, but if your application changes, your test will still need to update. The automation is just an implementation of a user flow. You wouldn't want your source of truth to be how your code is implemented because bugs exist.

We use browser drivers as a mechanism for interacting with the browser itself, but we don't generate code. We don't generate Selenium or Playwright code. What you see in your test definition—the natural language—is translated at runtime into an interaction in the browser. That's how we maintain very high levels of reliability and, surprisingly, speed. In most of our customers' benchmarks, we're on par with Cypress specs and about 20% faster than Playwright tests.

A core component of our approach is that we don't generate Playwright code, and that's by design. If you generate Playwright code, you still run into the same problem—you still have to maintain it, and that's the bigger cost long-term.

If everything goes right for Momentic over the next 5 years, what does it become and how is the world changed?

Where we're starting right now with web testing is just our entry point. If you think about testing, it's a huge field—we're just a tiny slice of the pie with web end-to-end testing. There are many directions we can go, and five years is a lot of time for a startup.

There's security penetration testing, accessibility auditing, user acceptance testing (is this feature intuitive for customers?). There are other parts of the stack close to end-to-end testing, like API testing. For many customers, their primary product is an API, so API testing is essentially their end-to-end testing.

The testing world is quite large. There's also mobile, desktop, Windows applications—many things that need testing. Our vision is to become that one tool that, when you think about testing, Momentic is the one that will help you test, regardless of what you need to test.

Disclaimers