Low-Code Platforms in the Age of AI

A JFDI Whitepaper

In a world where anyone can write running code in minutes by describing what they want to an AI, who needs clunky low-code platforms any more? Or software engineers? Or even system architects for that matter?

When the need for business applications outstrips the availability of the human resources to satisfy them, AI seems to provide all the tools we need to make the problem go away.

Too good to be true? JFDI's Director of Research and Development, Jon Silver, explores the AI-dominated development landscape in the last half of the 2020s, and the ups and downs of where software development will go from here.

JFDI's ***Locodium®*** *platform is also introduced - Jon discusses why it's different and how it fits into an increasingly AI-dominated world of software development.*

JFDI Consulting Ltd
Website: **jfdi.info**
eMail: **hello@jfdi.info**
Phone: **01273 358282**

Contents

Low-Code Platforms in the Age of AI

Some Essential Concepts Defined

First of all, it's important to understand some of the concepts we'll be discussing… so here’s our attempt at defining some basic terminology for you. But if you know this stuff already, feel free to skip ahead.

LLM - Large Language Model

A Large Language Model (LLM) is a type of machine-learning model that learns to understand and generate human-like text by studying massive amounts of written material; like books, articles, and websites. It’s trained in two main steps: first, it reads all this text and learns to predict what word comes next in a sentence, which helps it pick up grammar, facts, and even some reasoning skills; then, it gets extra training with human feedback to make its answers more helpful and safe. As a result, an LLM can answer questions, summarise documents, translate languages, write stories or emails, and even help with coding; essentially acting as a super-smart text assistant that can handle almost any language-related task. Although highly capable, LLMs do not possess genuine understanding or consciousness — they simulate it through pattern recognition.

Context Window

A context window is the amount of information — like words, sentences, or data — that an AI model, such as a Large Language Model, can “see” and consider at one time when generating a response or making predictions. Think of it as the AI’s short-term memory: if you give it a long document or conversation, it can only pay attention to a certain chunk at once, and anything outside that chunk is ignored. The size of the context window limits how much detail or history the AI can use to understand your request, answer questions, or keep track of ongoing discussions, so bigger context windows allow for more complex and coherent interactions. So far the largest context window any LLM has had is a million tokens - roughly 750,000 words, or 1,500 pages of 500 words per page - which sounds a lot but is quickly filled when executing coding tasks with a lot of "context engineering" content.

Vibe Coding

"Vibe Coding" refers to the practice of using AI tools — especially Large Language Models (LLMs) — to quickly generate code by simply describing what you want.

Context Engineering

Context engineering is deciding what information to give an AI so it might produce useful code. You can spend hours carefully organising documentation, existing code and requirements, but the model will still randomly ignore critical details, hallucinate solutions, or produce code that doesn't work with your existing system.

The term "engineering" makes it sound precise when it's really just "give the AI more stuff and see what happens". It's basically informed trial-and-error. Better context improves your odds but guarantees nothing. The AI might focus on irrelevant details while missing the crucial constraint buried in your specification. Sometimes more context makes things worse by confusing the model or hitting token limits.

Agentic Coding

Agentic coding is software development carried out by AI agents that plan, write, run, and revise code in iterative loops. Unlike simple code generators, these agents carry context across steps within a session, use tools such as compilers and debuggers, and adapt their approach when errors occur.

Its appeal is faster handling of routine development and attempts at autonomous problem-solving without constant human input. But major limits remain: restricted context windows, no true memory across sessions, inability to grasp business or architectural intent, repeated failed strategies, difficulty with complex builds, and poor coordination in multi-agent settings.

Low-Code Platforms/Tools

Low-code platforms are software environments designed to enable rapid application development through visual interfaces and prebuilt components, minimising the need for traditional programming. They allow users — often without any software engineering experience — to assemble workflows, integrate systems, and deploy functional applications by configuration rather than code. While marketed as democratising software creation, these tools frequently abstract away essential complexity, encouraging shallow solutions that are easy to prototype but hard to scale, secure, or maintain. Low-code platforms trade long-term robustness for short-term convenience, producing brittle, ill-fitting systems that struggle under real-world complexity. This leaves organisations locked into expensive subscription models and constrained extensibility, rather than benefiting from genuine engineering discipline.

More Software Needed

The world needs more software written than ever before. The sheer number of places where some sort of computer does some sort of automation, has increased exponentially over the past two decades. Software has always been written by people - programmers, developers, or software engineers. They can write software as fast as they can think & type, but there's a limit. So we either need more skilled software engineers or we need to make the software engineers we already have more productive. Low-code tools and platforms are one way to do this, either by enabling skilled developers to work faster, or by taking small systems away from them and into the hands of users. But AI now presents a huge range of ways to speed up the development process, which we'll look into shortly.

So you might think that's it: between these two areas of technology, we must have cracked the software engineering velocity barrier. The answer to that assertion is, as so often, nuanced and varied across industry.

LLMs as Coders

"Vibe Coders" use LLMs to generate traditional code. Generally it's considered more productive to do this via some sort of agentic coding tool like Anthropic's *Claude Code*, Open AI's *Codex* or the independent *RooCode*, which aim to aid the human user in Context Engineering - providing additional context to the LLM through existing code, documentation and standing instructions - and even splitting up the task into subtasks that different AI agents can tackle.

Code generation is nothing new as a concept – it's been used for decades, starting in a more innocent age with self-modifying machine code. Code generation is how some of the no-code/low-code tools still do their thing, except they do it using templated blocks of pre-written code.

The term 'engineering' makes it sound precise when it's really just 'give the AI more stuff and see what happens'.

On the face of it, there's nothing wrong with getting an LLM to code for you. After all, they've seen a lot of code, because they've been shown a lot of code. But all the code they've seen is publicly available and anonymously viewable. However, not all extant code is equal, and publicly available code is often of poor quality. You can probably guess why this is a problem, but before I explain it fully let's look at an old cornerstone of computing.

Although the modern expression was apparently first coined in 1957, the concept of "garbage in, garbage out" (GIGO) dates back to the original pioneer of computing, Charles Babbage, in the 19th century.

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

Thus it's long been understood that in any system, the quality of the output depends heavily, if not entirely, on the quality of the input. This quality requirement never went away, and is as relevant to code generation through token-prediction as it was to computation through the meshing of cog wheels.

Better context improves your odds but guarantees nothing.

Where do LLMs Get Their Knowledge?

So, returning to the training data of the current generation of LLMs... a lot of the code they're trained with isn't very good code. Thus, unless you know exactly what you're doing, and can appraise the quality of the code being churned out by your LLM, and further hone and guide your prompts to ensure that the output is of excellent quality, then what you're producing is tons and tons of AI "slop" code.

Consequently, there's a massive problem that's waiting for us just down the road; I'll return to this shortly. But for now let's stick with the local problems that your inexperienced Vibe Coder is going to be building up.

Business systems - line-of-business applications, or whatever you call an entire suite of business automations - are complex things. There are many moving parts in any modern business app, built with many layers, each serving its own set of purposes. They're hard enough for an experienced system architect to keep track of, let alone an LLM with a strictly limited context window.

The Burden of Ownership

When you develop software, write code, program a computer - whatever you want to call it - you're writing lines of instructions. Every line of code becomes a permanent part of your system; a component part of a business asset. So you'd think if it works right now, it'll always work... right? Well, yes and no. From the moment it's written, the clock is ticking. The code becomes stale over time, because the language evolves and the libraries of external code it's calling change. Without strict version control over tools and libraries, that code could be out of date within a year. Luckily, we have mechanisms built into most development environments that will allow us to tightly version libraries used during the build process, so our old code will always still work. But at some point there'll be a change somewhere in the dependency tree, due to a security threat or upstream change, that mandates the updating of one of our libraries. If your old code was using a now-obsolete function signature, it now won't work and needs updating.

Thus, there's a fundamental principle here: *every line of code is a maintenance burden*.

Right now there are more people producing a greater volume of code faster than ever before. Vibe Coders are publishing more and more of their "look what I did in just a few days with ChatGPT!" repositories on GitHub every day. Reddit is a great place to see many such announcements.

The Danger of Amateurism

All that poor-quality code produced by LLMs, overseen by people with no development experience and no idea how to improve it. The problems don't stop at maintainability either. A lot of the AI-generated code I've seen first-hand is full of potential security holes, and is absolutely unsuited to deployment on publicly-accessible production servers. Yet there are startups being founded on the basis of the software their IT-illiterate rookie CEO vibe-coded with an AI in a few weeks at home.

The Tea App disaster stands as the archetypal cautionary tale of why vibe-coding represents an existential threat to user safety and data privacy. This women's safety app was founded by Sean Cook, who reportedly had limited coding experience beyond what some sources claim was a six-month bootcamp. In July-August 2025, it promptly suffered three catastrophic data breaches that exposed 72,000 images including driver's licenses and government IDs, plus 1.1 million private messages discussing deeply personal topics like abortions and abuse.

All of this happened because Cook apparently outsourced development to Brazilian contractors while embracing a hands-off "trust the vibes" approach that allegedly left fundamental security practices like data encryption and access controls completely ignored. Women's most sensitive information was leaked on 4chan, where misogynistic users mocked and threatened their safety. This wasn't sophisticated hacking — it was what appears to be criminally negligent architecture where databases sat wide open on the internet. A non-technical founder seemingly believed AI-assisted coding could substitute for actual cybersecurity expertise, ultimately proving that when tech bros prioritise "moving fast and breaking things" over protecting vulnerable users, they literally break people's lives.

Code Overload

And there's another LLM habit we should look at for a moment - unless told to use a specific library to improve maintainability, safety and reliability, LLMs will tend to code everything from scratch, producing thousands of lines more code than a professional software engineer would. The volumetric absurdity doesn't stop there either... they'll often write the same code repeatedly with no reuse, and poor modularity. LLMs aren't even very good at looking at what's already available in the existing codebase and reusing that... they'll happily go on reinventing poorer-quality versions of wheels over and over again.

Every line of code we write is a line of code we'll have to maintain.

Right now there are more people producing a greater volume of code faster than ever before.

So much code in so little time. So little code quality. Little or no adherence to best practices. So few checks and balances. Years worth of technical debt. So much showboating.

But it's ok, you might say. LLMs will improve. All technology improves over time. Doesn't it? The code produced by LLMs will become better. LLMs will come along that can sort out the mess for us.

And here we come to that massive problem that I mentioned earlier. AI-generated code is usually of poor quality, but produced thousands of times faster than a human programmer could write it. Alas the vast quantity of low-quality AI-generated slop code being churned out and published into publicly available repos will end up in the training data for the next generation of LLMs.

The sheer quantity of AI slop code entering the public domain threatens to overwhelm and dilute the influence of all the human-produced code that's of any quality at all. *See the problem?* This isn't a situation where the AIs can help us out of the quagmire created by an earlier generation of AIs - this is a rapid, global, iterative, exponential reduction in the ability of AI to be at all useful as a tool for code generation.

Meanwhile, it seems some C-suite execs are being taken in by the magic of AI. Look at the balance sheet improvements we can make if we sack our expensive, slow, argumentative developers and have our managers produce all the code we need using AIs! Brilliant!

These ticking timebombs are all out there waiting to implode any organisation willing to get too close. This isn't made up. It's all real, and happening now.

When tech bros prioritise 'moving fast and breaking things' over protecting vulnerable users, they literally break people's lives.

Conceivably someone could produce a beautifully trained LLM, targeted specifically at adhering to all known best practices, able to spot every anti-pattern, and produce perfect code. It would have to be a very large model (GPT5/Claude Opus 4.1 sized) to be sufficiently effective, and its training data would have to be hand-picked as a set of perfect examples. Then perhaps we could let it loose on a miscreant codebase and have it right all the wrongs, and write new code that didn't merely proliferate poor practices. Does such a model exist? Is anyone working on such a thing? No, not as far as we know. Is it possible? In theory, yes, but that training data would be the real challenge.

So what's the answer? Could it be AI-Assisted Development?

AI-Assisted Development

So far I've only talked about the downsides of AI development done as "vibe-coding". It may therefore come as a great surprise to you to learn that we use LLM-powered tools here at JFDI to write a lot of our code. Why wouldn't we? I started out 45 years ago writing machine code as binary-encoded hexadecimal - if I wrote business applications that way, I'd still be coding my first one. Our tooling, techniques, languages and methodologies have all evolved over time. Anyone who doesn't keep up with all of that will get left behind.

The thing is, we have the experience to know how to use those tools wisely and effectively to produce the high-quality code we need, for optimum maintainability, reusability, safety, reliability and security. Sometimes it takes many iterations. Lots of swearing at LLMs too. But the end-product is new features added to software at record speed - in days rather than weeks, or weeks rather than months.

This is AI-assisted development. It's not perfect. It's hampered by the tendency of the LLMs to revert to type, producing their slop based on all the low-quality examples they've seen. When you tell them to adhere to a pattern when it's necessary, they can obsess about using that pattern everywhere regardless of the need - for example inserting error-trapping code where no error will ever occur. We have to be ultra observant, conduct on-the-fly code reviews, make manual changes and not let any thought process go off the rails.

So you'd think that this might be the way forward. And it is a good way, at least for now. Except there's not a lot of development teams as experienced as JFDI's, and certainly none in any startup or corporate IT department we know of. It's not that corporate IT people are deficient - merely that most of them exist in a technology bubble that doesn't change much, so they tend to suffer from knowledge obsolescence and skills lag once they've been in the job for a few years.

The sheer quantity of AI slop code entering the public domain threatens to overwhelm and dilute the influence of all the human-produced code that's of any quality at all.

There's lots of software engineering experience in software houses, consultancies and so on, and if they're not doing software development in 2025 the same way we're doing it, using the currently available tools to get a 5x speed increase, then they're probably facing extinction.

But however fast professional software engineers can go, what none of this will do is to put commercial-quality software authoring into the hands of ordinary users.

And that's exactly what low-code platforms and tools were supposed to do. So what went wrong?

Low-Code Application Platforms and Tools

Through a process of reduction, one can write out a list of the basic building blocks that need to be included in every possible business system. For example reporting, or user notification, or long-running processes involving various data and some occasional human input. It sounds good, doesn't it? Opening up the possibility of assembling these generic building blocks into any shape to satisfy the requirements of any business system. In effect, creating the ultimate generic abstraction of the business process, so it can be shaped and specialised to fit any business. However as so often is the case, it's easy to say the words but extremely hard to deliver on that idea. And lots of companies have tried and failed, or tried and produced something that sounds great on paper but isn't that great to use.

Imagine that most fundamental element of the business system building blocks, the data-entry form. Fundamental, yes, but full of complexity. Each data-entry component on every form must look identical and work consistently with all the others. Each form you create must work the same way, have the same basic facilities, but represent a different set of business data with all its constraints and validations. A form is the first line of defence against out-of-range, badly formatted or mutually inconsistent data. But it also needs to be easy to use and not cause friction or frustration for the user.

Hand-coding all that is a massive task. And yet that's how most systems are still created. Yes, they might employ components that encapsulate the desired behaviours and make in-place validation possible, but nonetheless all the forms in most systems are mostly hand-coded. In other words, fields like first name, surname, job title and so on all have corresponding blocks of program code somewhere, representing how they're displayed, entered into the form, and validated.

Low-code application platforms and tools seek to abstract out all that software engineering complexity, and perhaps give the user a nice canvas for creating forms. Sounds lovely - idyllic almost. Except visual form creation is still a slow, arduous process on all the clunky, resistive visual editors we've ever used – which pretty much describes all of them. The main problems invariably arise when you want to do anything dynamic - for example calculating one field value based upon others, or validating a set of fields based upon the data from elsewhere on the same form. You know, real-world business complexity. At that point, most low-code platforms make you drop back to good old fashioned code, which users are ill-equipped to do.

This is very much the case for Appian, for example, where its proprietary SAIL language ends up being the catch-all escape hatch for every scenario the platform can't manage through its visual user interface – which is seemingly most of them. The use of visual interfaces, and in particular the inability of low-code platform vendors to create a user interface that is both simple and powerful enough to rapidly describe their business system requirements, severely limits the ability of these platforms to deliver on their promise.

Low-code platforms trade long-term robustness for short-term convenience.

They produce brittle, ill-fitting systems that struggle under real-world complexity.

Another type of low-code platform allows you to visually create a diagram of entities, relationships and data flow, and then generates actual code that you run through a traditional compile and build process. This is a truly terrible approach – the worst of all worlds. Although it may be perceived as producing something real and tangible - insurance against obsolescence or the discontinuance of the low-code platform - what you're in fact left with is a vast quantity of someone else's code, of variable quality, and the burden of maintaining it. The code-generation process is generally one-way, so modifications to the code are a dead-end, and out of warranty. And if the generated code employed code block templates containing bugs, those bugs are now replicated over and over again throughout your business system. And as I've pointed out, the new generation of AI-based code generators doesn't really help get over these fundamental problems.

Of course low-code apps don't stop at forms. And nor do the restrictions and roadblocks imposed by the low-code tools and platforms.

JFDI's Locodium® offers full-scale business applications with no compromises, in days not months, or weeks not years.

Disappointments and Disillusionment

Low-code and no-code development platforms promised to democratise software creation and accelerate digital transformation across industries. From 2019 to 2024, organisations of all sizes – enterprises, SMEs, government agencies, and non-profits – invested in these tools with hopes of quick wins and reduced dependence on scarce developer talent. However, many adopters have reported underwhelming outcomes, with projects failing to deliver the expected success. Key shortcomings observed in practice include limited scalability, security and compliance gaps, maintainability issues, governance challenges, integration obstacles, user disillusionment, vendor lock-in, and hidden long-term costs.

Notably, initial prototypes often cannot scale to enterprise-grade deployments, forcing costly rewrites in traditional code once an application grows beyond a certain point. Security and governance weaknesses have emerged as serious concerns, especially in highly regulated sectors where citizen-developed apps introduced vulnerabilities and compliance risks. Maintainability and technical debt have proven problematic – the speed of visual development often comes at the cost of sound architecture, leading to brittle systems that are hard to debug or extend. Integration with legacy systems is another pain point: many no-code solutions operate in silos and struggle to interface with existing databases or services, resulting in fragmented processes.

Furthermore, the much-touted "citizen developer" revolution failed to fully materialise. Business users encountered steep learning curves and platform limitations, causing frustration when they hit a complexity ceiling or needed support from IT for non-trivial tasks. In many cases, IT departments had to step back in to provide training, oversight, and to retrofit proper governance once shadow IT app sprawl became evident. Meanwhile, professional developers often criticise low-code tools for lack of flexibility and control, finding them unsuitable for complex, mission-critical software and worrying about accumulating technical debt. This has led some organisations to dial back citizen development initiatives in favour of blended teams or more controlled approaches. Indeed, by 2024 there was a growing recognition that low-code is "not a silver bullet" and must be adopted with careful planning to avoid being "penny-wise, pound-foolish" in the long run.

In summary, while low-code/no-code platforms delivered successes in niche areas (especially for rapid prototypes and simple internal apps), they often disappointed in broader enterprise use. Many promised benefits – faster delivery, lower costs, empowerment of non-IT staff – were offset by unforeseen challenges in scalability, security, maintainability and total cost.

Low-Code vs AI Alignment Dangers

As we've discussed already, if you give the keys to the kingdom to an AI, you should expect bad things to occur. If you allow an AI to write all the code, including all the security and authorisation features, you invite disaster.

But if you limit what the AI can do, for example only allowing it to configure a low-code platform rather than writing all the code, and you severely limit its ability to do damage. At the same time every configuration the AI adds, and every customisation the AI-assisted human adds, leverages the exhaustively tested building blocks that comprise the foundations of the platform, which in turn encapsulate all the experience and wisdom and hard-earned lessons of a battle-hardened human software engineering team.

So what if there was a low-code platform that didn't choke on serious business applications? What if it was accessible enough to low-end developers and end-users, but sufficiently powerful to create full line-of-business applications? And what if that platform was designed from the ground-up to work with Agentic AI to multiply output velocity by 5x to 100x?

Meet Locodium® from JFDI.

Locodium® - the Modern Antidote to Low-Code Platforms

At JFDI we've long sought to rid ourselves permanently from the need to code another set of data-entry forms ever again. It's boring, tedious, repetitive, detailed and annoying. It adds a disproportionately long chunk of development time to any system project, making us slow, extending product delivery timescales, and detracting from the resources we can direct at business logic and process automation. Over the years we've individually and collectively refined our thinking about how we could best do this - always within the restrictions of the background building block technologies available to us.

What if forms provision was just taken care of and made possible by the platform? What if one's data definition, the shape of the database, could just be used to generate the form, and perhaps have some extra formatting, dynamic calculation or validation built in? What if there was a single, simple language that could be used to express all this dynamism, not unlike expressions in a spreadsheet like Excel?

Well, that's exactly what we did when we created our forms engine, Formulationist®. Formulationist® is a central component of our Locodium® applications platform, and when you develop a Locodium® app you'll be using a whole bunch of Formulationist® forms and other bits of Locodium®. There's also LWE, the Locodium® Workflow Engine, and a whole host of other JFDI's homegrown technologies.

We created Locodium® to be a platform driven by data and metadata, but truly without limits. There are well-defined interfaces for extending Locodium® in any way we want. The definitions of forms, data entities, pages, workflows and all the other Locodium® business objects are written in small JSON files (a standard text-based data definition language) - so whenever an edit would be too slow in any visual editor, we can edit the definitions directly. But even better, our definition schemas are so easily understood by LLMs that we can use them to generate whole swathes of Locodium® configuration.

In this way, Locodium® truly is the best of all worlds: a low-code platform written differently from all others, to transcend all the limitations of all prior low-code platforms, but driven by small, modular definitions that can be generated rapidly and en masse by the current and future generations of LLMs. Not only is there the escape hatch of dropping down from a visual editor to a guided text-based one, but also the ultimate escape hatch that any new extension to Locodium® can be easily added in by a developer and immediately made available to all apps in the tenancy. If there's a bug in the platform code, its fix will reach all apps in the tenancy as soon as the platform software is redeployed.

The result is app development at huge velocity - think 50x or 100x. But unlike vibe-coded or agentically-coded apps, these apps all have the same basic quality built in, including role-based security down to row level, adherence to security best practices and more.

Locodium® centralises all these foundation elements of business applications:

- Security: Providing granular access controls across multiple dimensions, including user-level, group-level, entity-level, row-level, and field-level, to protect sensitive business data and enforce organisational policies.
- Role Management: Enabling dynamic assignment and delegation of roles, permissions, and responsibilities, crucial for flexible and efficient operational management.
- Forms: Robust data entry mechanisms featuring intricate validation rules, embedded business logic, and seamless integration with both existing and newly structured datasets.
- Workflows: Supporting complex, long-running business processes involving automated data ingestion, transformation and filtering, manual human interventions, and interactions with external and internal APIs.
- Data Management: Ensuring secure, reliable storage, sophisticated versioning mechanisms, and rapid retrieval of files and structured data records.
- Dashboards: Offering real-time analytical insights through highly customisable dashboards, interactive widgets, and comprehensive reporting functionality.
- Notifications: Facilitating event-driven communication channels, including email, SMS, in-app notifications, and webhook integrations, to ensure timely awareness of critical business events.
- Integrations: Seamless connectivity with external services, databases, and third-party APIs, enabling cohesive integration with existing organisational ecosystems.
- Auditing: Comprehensive logging and traceability of system activities to support regulatory compliance, accountability, and forensic analysis.
- Scheduling: Advanced time-based triggers, automated reminders, and integrations with existing calendar systems to streamline time-sensitive business processes.
- Localisation: Full support for multi-language environments and regional customisation of data formats and user interfaces.
- Mobile Access: Responsive user interfaces and flexible mobile deployment options, enabling business-critical functionality on any device.

- Customisability: Extensive tailoring options, white-labelling capabilities, and extensibility through configurable modules and open interfaces.

There are many low-code application development platforms and products. But there is none like Locodium®. The most successful apps we've seen developed on other low-code platforms are mere toys, lightweight and throwaway – like an app to collect office workers' lunch orders, or to garner peoples' opinions on matters of workplace planning. They certainly wouldn't be able to produce a complete LOB (Line Of Business) system automating every aspect of the business process, yet Locodium® can – and in a mere fraction of the time it would take to produce such a system using traditional development tools and languages. Locodium® represents a true leap of innovation, the culmination of many years of R&D innovation, and the confluence of many of our innovation outputs.

Locodium® offers full-scale business applications with no compromises, in days not months, or weeks not years.

About JFDI Consulting

Company Background

JFDI Consulting Ltd was founded in 2001 to provide IT systems consultancy and software development services to the SME market. Our mission was then, and still is, to utilise all available contemporary technologies to deliver measurable improvements to our clients' business processes.

New technologies appear all the time, with a wave of aspiration as to their potential benefits to business and the world; but without being joined together with other technologies, adapted and applied to real-world problems, they remain merely interesting curiosities.

Meanwhile, our clients exist within highly competitive markets, and their ability to do whatever they do faster, more accurately and at lower cost is key to their continued survival and growth.

JFDI specialises in providing solutions for business problems by finding the best technologies for the job, and creating systems that incorporate those technologies. We're a small company, so in order to survive and thrive in our own competitive market we must address the unsolvable problems: those that others have thus far failed to solve, whatever the reason, whether it be high cost, technical inability or the lack of requisite foundational technologies. We can only do this by investing heavily in our R&D efforts, to identify potentially useful technologies and learn how to harness them to create the most innovative, leading-edge solutions for business challenges. As a result, JFDI has created an abundance of bridge technologies, proprietary platforms and tools that our clients continue to benefit from, as we bring to bear the very best in current technologies to help them do what they do, better, faster and with higher fidelity than ever before, giving their users true superpowers. Faster, with multiplied productivity, with 100% accuracy and effortless repeatability.

JFDI Consulting Ltd
Website: **jfdi.info**
eMail: **hello@jfdi.info**
Phone: **01273 358282**

About the Author

Jon Silver is co-founder of JFDI Consulting with Joel Jeffery. He has over 40 years of experience in various branches of engineering including systems architecture, software development, microelectronics design and associated engineering disciplines, and has designed and implemented solutions across many industries. He has been considered an authority in software authentication technologies and database technologies as well as Rapid Application Development, speaking on related topics at many conferences, and has been a writer and contributing editor for a number of internationally published technology journals, and contributor/editor on several technical books. Jon has seen many technological paradigm shifts, and how early promises for the future often fail to materialise once the hype has died away. He began inventing as a child (for example, the "Sprayless Mudflap" for cars and lorries), and has never stopped innovating. Some of his contributions from thirty years ago are still in use today within cloud services used by millions of users every day. He strives for a deep understanding of every aspect of science and engineering. He uses the phrase "standing on the shoulders of giants" to describe the synthesis of invention, acknowledgement of the cumulative nature of human innovation through a multiplicity of layers of technology applied to problems over time to create the human world of today, and indeed of tomorrow.