The Hidden Cost of AI Application Delivery

A Technical Debt framework for adopting AI without multiplying software liabilities

Core thesis

AI makes applications easier to create. It does not make them easier to operate, govern and maintain. The organisations that win the AI adoption mandate will not generate the most code; they will build more capability while carrying less lifecycle burden.

CEOs, CIOs, CTOs, COOs, transformation leaders, business-unit executives, technology partners and delivery organisations accountable for using AI to improve speed, efficiency and competitiveness without creating unmanaged technical debt.

Contents

Executive summary

Almost every organisation now carries a version of the same mandate: use AI to become faster, leaner and more competitive. That pressure is rational. AI can now turn ideas into working applications with extraordinary speed. But a working application is not the same as a governed, secure and maintainable system that can operate inside a real business.

The strategic risk is that organisations mistake time-to-demo for time-to-value. Every application created by an internal team, an external partner or an AI coding tool becomes part of the organisation's software estate. It must be secured, tested, patched, documented, upgraded, integrated and evolved. Those obligations are the hidden cost of AI application delivery.

This paper uses a practical Technical Debt Index to show where that cost accumulates: structurally, operationally and organisationally. It then shows how Buzzy reduces that debt by minimising per-application custom code, using a governed semantic application layer, reusable platform-maintained capabilities and a core engine that improves every application at once.

> **Leadership takeaway**
>
> The question is no longer whether AI can help build software. It can. The question is how much long-term software burden the organisation is prepared to carry as AI makes application creation easier for every team, vendor and business unit.

The paper in five points

- **Creation is no longer the scarce part.** AI has compressed the path from idea to demo. That is valuable, but it is not the same as durable business value.
- **Technical debt is multidimensional.** It appears structurally in code, architecture and data models; operationally in security, testing, CI/CD and uptime; and organisationally in documentation, handover, vendor dependency and product lifecycle drag.

- **Technical Debt Ratio reveals the trap.** AI can lower the apparent build cost while increasing the long-term remediation and maintenance cost, making the ratio worse over the lifecycle.
- **Buzzy changes the lifecycle economics.** Instead of generating a bespoke codebase for every app, Buzzy creates a semantic application definition that runs on a centrally maintained core engine.
- **The buyer needs a board narrative.** The value is not simply “less code”; it is a governed way to adopt AI, digitise workflows and scale application delivery without creating a fragmented software estate.

External signal: technical debt is now a management category

Market validation, not a Buzzy claim. Gartner now treats technical debt management as a formal software category, with tools that analyse source code, architecture and dependencies to identify, visualise and prioritise technical debt, structural flaws and security risks. Gartner projects architectural technical debt will account for 80% of all technical debt by 2027 and estimates the technical debt management tools market will reach $1.2B in annual revenue by the end of 2026, with double-digit growth through 2030. The nuance matters: Gartner also reports that 39% of engineering leaders and teams noticed AI reducing technical debt in newly generated code, while broader AI coding adoption is increasing architectural debt across systems. Gartner is cited for market context only and does not endorse Buzzy.

Source note: *Gartner, Magic Quadrant for Technical Debt Management Tools, 20 May 2026; DX, Technical debt ratio: How to measure technical debt, updated 28 Oct 2025; vFunction, How to Measure Technical Debt, 11 Jul 2024.*

1. Time-to-demo is not time-to-value

The first wave of AI coding changed expectations. Business teams, founders, product owners and service leaders can now watch an AI tool produce a working interface in hours. That speed is real, and it will reshape how every organisation thinks about process improvement, customer experience and internal automation.

But a demo optimises the wrong measure. A working application is the start of the organisation’s obligations, not the end of them. The day an application starts to handle customers, staff workflows, financial data, health data, operational decisions or compliance evidence is the day it must be governed like any other business system.

The hidden cost sits in everything that happens after the first working version: security, compliance, data protection, dependency patching, framework upgrades, monitoring, incident response, testing, documentation, integration, user support and change requests. These costs are not incidental. They are the long tail of software operation and maintenance.

The trap

AI makes it easier to create more applications. Without an architecture for reuse and governance, it also makes it easier to create more codebases, more suppliers, more dependencies, more unpatched vulnerabilities and more systems that nobody fully understands or governs.

2. A practical lens: Technical Debt Index and Technical Debt Ratio

Technical debt is often discussed as a vague engineering concern. That framing makes it hard for boards, executives and business leaders to act on it. A practical Technical Debt Index makes the liability visible by separating it into three dimensions: structural, operational and organisational debt. This aligns with the direction of the emerging technical debt tooling market, which increasingly combines code-level analysis, architecture observability, security signals, scoring and financial impact modelling.

Technical Debt Index dimensions

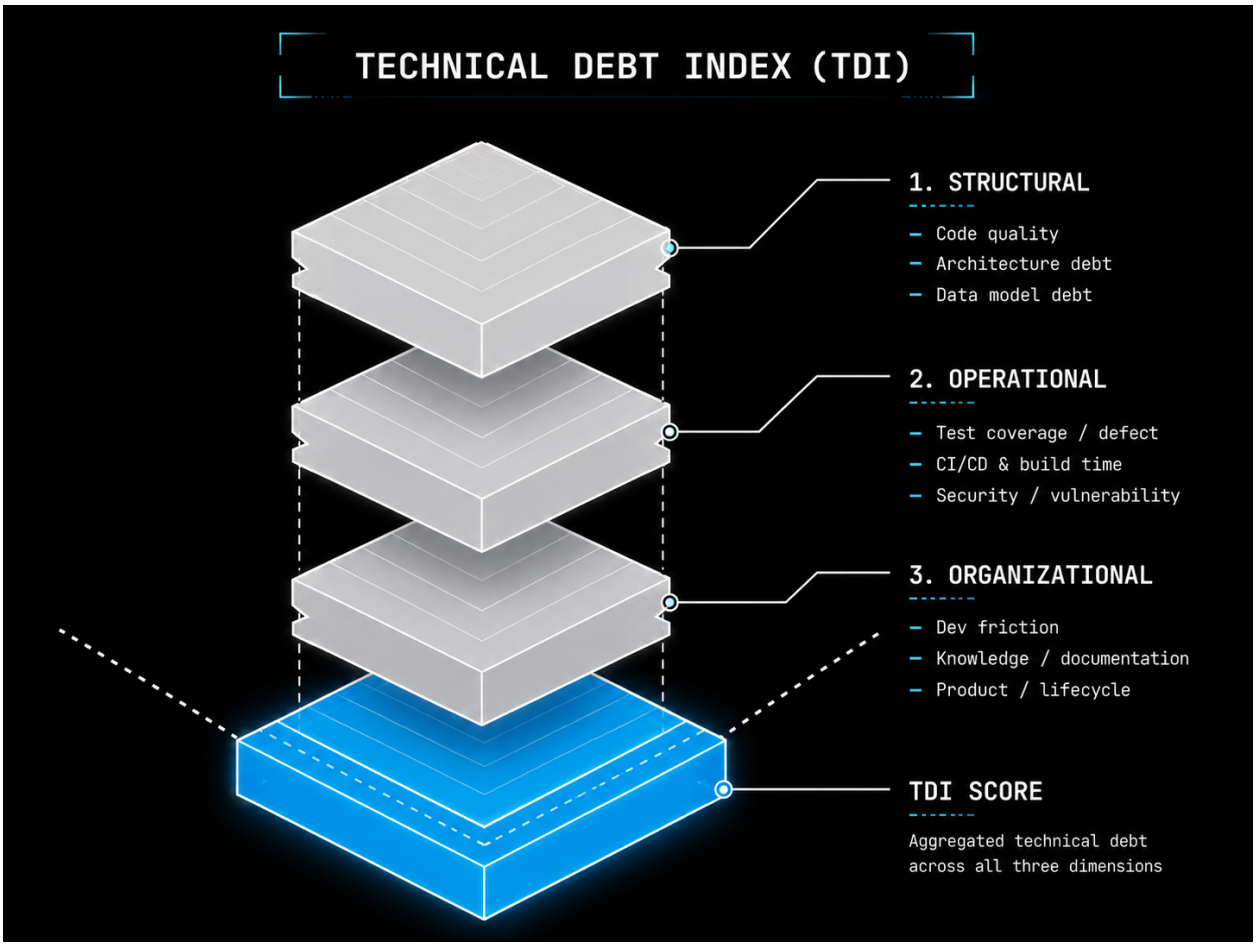


Figure 1. Technical Debt Index (TDI): structural, operational and organisational dimensions.

TDI dimension	What it means	Why executives should care
Structural debt	Code quality, architecture debt and data model debt.	Poor structure makes every future change slower, riskier and more expensive. It also increases dependency on whoever originally built the system.
Operational debt	Test coverage, defects, CI/CD, build time, security vulnerabilities, monitoring and release discipline.	Operational debt becomes downtime, incidents, audit issues, delayed releases and unplanned remediation work.
Organisational debt	Developer friction, knowledge gaps, documentation weakness, vendor dependency and product lifecycle drag.	Organisational debt shows up as slow handovers, expensive change requests, loss of institutional knowledge and portfolio-level complexity.

Technical Debt Ratio: the commercial lens

For executive use, Technical Debt Ratio can be treated as a lifecycle-cost signal: how much future remediation and maintenance burden the organisation is accumulating relative to the apparent cost of creating the software.

$$TDR = (Remediation\ Cost / Development\ Cost) \times 100$$

External threshold framing makes TDR useful for executive discussion. DX describes TDR as (Remediation Cost / Development Cost) x 100 and suggests teams below roughly 5-10% generally maintain healthy velocity, while ratios above 20% typically signal systemic issues requiring strategic intervention. vFunction separately describes a minimal TDR of less than 5% as ideal. These are directional thresholds, not universal rules, but they help separate manageable debt from debt that is likely to constrain delivery.

In practical terms, the ratio is a commercial warning signal: AI generation can reduce the apparent effort to build while increasing the downstream effort to fix, secure, test, document, release and evolve the application.

The implication is simple: the goal is not merely to reduce the cost of generating software. The goal is to reduce the debt that remains after the software exists.

3. Where AI-generated applications create hidden debt

Raw AI-generated code is often evaluated at the point of creation. A more useful evaluation is to ask what it adds to the Technical Debt Index over the life of the application. The issue is not that AI code cannot work. It can. The issue is that every generated codebase becomes another asset the organisation must govern, secure and evolve.

Debt driver	How raw AI-generated code can increase it	How Buzzy reduces the exposure
Code quality	Generated implementations may duplicate logic, hard-code assumptions and vary in naming, error handling and edge-case coverage.	Common application patterns run on maintained components instead of being regenerated per app.
Architecture debt	Each app can receive a different architecture, dependency set and integration pattern, even when the business patterns are similar.	A semantic app definition runs on a common core engine, enforcing a more consistent execution model.
Data model debt	Schemas, migrations, access rules and retention logic are recreated for each app and must be maintained separately.	The data model is represented in the app definition and executed through platform-maintained data capabilities.
Security, compliance and vulnerabilities	Every endpoint, access rule, validation path and compliance obligation must be implemented, evidenced and rechecked after change.	Buzzy includes built-in security controls and compliance assessment capability. Because the app definition captures data, permissions, workflows and intent, organisations can see, assess and control related capabilities and risks more clearly.
Testing and release	Each app needs its own test plan, pipeline, regression suite and upgrade discipline, often inferred after the fact from generated code.	Buzzy includes built-in testing capability. Because it understands the app definition, tests can be generated and run against the intended data model, flows, permissions and UI behaviour.
Knowledge and documentation	Understanding sits inside generated code, vendor habits and developer-specific decisions, so documentation is often created after the system is already built.	The brief, flows, blueprint, data model, theme/design system and requirements are part of the app definition. Documentation is inherent in the solution and drives the application, helping a new user or AI understand the system without reading raw code.

This is the key shift: the Buzzy comparison should not be framed only as “fewer lines of code”. It should be framed as a reduction in Technical Debt Index exposure across the structural, operational and organisational dimensions that drive long-term TCO.

4. The Buzzy model: composable applications on a semantic application layer

Buzzy is best understood as a governed semantic application platform, not simply as a code generator. It separates application intent from implementation burden. AI, Figma designs or requirements are used to create a semantic application definition: a structured model of the app’s brief, flows, blueprint, data model, theme/design system, screens, permissions, security and compliance assumptions, tests, integrations and behaviour. That definition runs on a trusted core engine that provides common application capabilities.

1. Business intent	2. Semantic app definition	3. Trusted core engine
Prompt, Figma design, requirements or workflow model describing what the organisation needs.	A durable, structured representation of brief, flows, blueprint, data model, UI, theme/design, permissions, tests, APIs and agentic flows.	Platform-maintained runtime for web, native mobile, data, security, APIs, MCP and lifecycle updates.

The app definition as the control plane

Because Buzzy has inside knowledge of the application definition, it can support security and compliance assessment, built-in testing, API exposure and agentic flows from a richer understanding of intent. The brief, flows, blueprint, data model and theme/design system are not after-the-fact documentation; they are part of the system definition that drives the application.

In enterprise architecture language, this is a composable model. Recurring application capabilities are packaged and maintained centrally rather than rebuilt in every project. The business still gets a specific application, but the organisation does not inherit a bespoke implementation of every common capability.

Composable capability	Traditional / raw generated approach	Buzzy approach
Forms, lists and dashboards	Generated per app; quality and behaviour vary by codebase.	Reusable native components maintained centrally.
Calendars, bookings and scheduling	Custom logic, edge cases and tests written per app.	Pattern-level component capability reused across apps.
Maps, search and filtering	Each app carries its own implementation choices and dependencies.	Common interaction patterns provided by the platform.
Security, compliance and access control	Hand-written checks, policy logic and compliance evidence must be reviewed per app.	Built-in security patterns and compliance assessment use the app definition to make capabilities, permissions and risks easier to see and control.
Web and native mobile	Separate build paths or additional codebases are often required.	One application definition can target web and native mobile.
APIs, integrations and agentic flows	API integrations and agentic flows are built and secured per application, often without a consistent control point.	Scoped APIs, integrations and MCP/agentic flows can be driven from the app definition, creating a stronger control point for exposed capabilities.
Upgrades and fixes	Patch, test and deploy each codebase separately.	Improve the core once and make the improvement available across apps.

Positioning language
Buzzy helps organisations build more applications while maintaining a much smaller bespoke code footprint. It gives AI-speed delivery without forcing every business unit, vendor or project team to create another bespoke maintenance estate.

5. Why this matters even when development is outsourced

Many organisations assume technical debt is less relevant when development is outsourced. In practice, outsourcing changes who writes the software; it does not change who remains accountable for the long-term consequences. The organisation remains accountable for security, compliance, data, uptime, customer experience, change cost and vendor dependency.

Outsourcing reduces delivery effort, but it does not remove Technical Debt Index exposure. Structural debt can sit in the vendor-created architecture and data model. Operational debt can sit in patching, testing, security, deployment and incident response. Organisational debt can sit in documentation gaps, supplier dependency, change-request dependency and the difficulty of moving to a new partner later.

Buzzy changes the outsourced delivery pattern. An external partner can still help define, design and deliver the application, but the output is not another bespoke codebase only that supplier understands. It is an application assembled on a governed semantic platform, with common capabilities maintained centrally.

Outsourcing punchline
Traditional outsourcing gives you delivery capacity. Buzzy gives you delivery capacity while minimising the bespoke maintenance estate that remains after delivery.

Outsourced delivery question	Traditional answer	Buzzy-enabled answer
Who maintains framework upgrades?	The vendor, the internal team, or a future support partner — usually per app.	The platform absorbs common upgrades centrally where they are part of the core.
Who understands the system after handover?	Knowledge may sit inside supplier-specific code and uneven documentation quality.	The application definition captures the brief, flows, blueprint, data model and theme/design system, giving new users or AI a clearer way to understand the system.
What happens when the vendor changes?	The customer inherits a bespoke codebase and must fund transition risk.	The app definition remains a durable description of requirements, structure and behaviour, reducing transition risk.
How are common capabilities reused?	Often recreated or customised per project.	As platform-maintained packaged capabilities.

6. What the CIO can tell the board

The board proposition should not sound like an engineering argument. Most executive peers already expect technology leaders to manage cost, quality, risk and delivery. The whitepaper needs to give the buyer a sharper internal message: this is how we adopt AI responsibly while improving business execution.

Board narrative

We can adopt AI to move faster without allowing every business unit, vendor or internal team to create unmanaged codebases. Our approach separates application intent from implementation burden. Business teams can move quickly, while common security, compliance assessment, testing, documentation, data, mobile, API, agentic-flow and lifecycle capabilities are governed through a shared application definition and centrally maintained platform capabilities. That gives us AI-speed delivery without creating a fragmented software estate that becomes more expensive and risky every year.

Stakeholder	What they care about	Buzzy message
CEO	Speed, competitiveness and execution.	We can digitise workflows and launch new capabilities faster without uncontrolled software sprawl.
CFO	TCO, hidden cost and capital efficiency.	We reduce recurring maintenance burden, not just the initial build effort.
COO	Operational reliability and standardisation.	Applications can support real workflows while remaining supportable and upgradeable.
CISO / Risk	Security, compliance, governance and auditability.	Built-in security and compliance assessment make app capabilities, data, permissions and risks easier to see and control.
Business units	Faster change and less dependency on scarce developers.	Teams can move from intent to application faster while staying inside a governed delivery model.
Board	Responsible AI adoption.	Buzzy provides a controlled way to scale AI application delivery without multiplying software liabilities.

7. Production evidence: where the maintenance burden did not materialise

A controlled build study is useful, but decision-makers will reasonably ask whether the model has survived contact with production. The strongest proof should therefore come before the lab comparison: real workflows and lifecycle evidence from production use.

Healthcare coordination platform — aged-care workflow

Context: compliance-sensitive aged-care coordination across web and mobile.

OneTap — QSR customer feedback

Context: offline-capable feedback and operations system for multi-site restaurants, with reporting integration and field reliability requirements.

Evidence: moved from 100+ Figma screens to a base product in approximately four days; web, iOS and Android from one foundation; single-tenant deployment for compliance needs; 30+ patches and three major releases in six weeks. Why it matters: the team focused on product and customers rather than maintaining a large bespoke codebase.	Evidence: more than 1 million surveys processed; Tableau integration; 20+ major React Native releases absorbed; 350–650 developer hours avoided on mobile alone; US$50k–$100k+ saved on React Native maintenance alone. Why it matters: Buzzy absorbed platform evolution that would otherwise become recurring mobile and backend maintenance work.

Evaluation questions about production evidence

Production references rarely map perfectly to every country, sector or operating model. A useful evaluation is whether the case evidence demonstrates the same lifecycle demands: workflow complexity, governance, data sensitivity, release cadence, integration needs and the ability to evolve without accumulating unmanaged debt.

Question	Useful evaluation lens
Has the platform been used in our country?	Look for the required deployment model, data-residency posture, governance controls and support approach rather than country name alone.
Has it been used in our industry?	Look for comparable workflow complexity, compliance sensitivity, data handling, mobile/offline requirements and pace of change.
Can the evidence translate to our application portfolio?	Look for repeated application patterns that can be governed and improved centrally, rather than many separate bespoke implementations.
What should we learn from early production cases?	Focus on lifecycle signals: time to launch, release cadence, maintenance avoided, testing and compliance visibility, and how clearly the app definition documents intent.

8. Controlled benchmark: same application, two lifecycle approaches

The ShortStay build study remains valuable, but it should be positioned as supporting evidence rather than the whole story. The study built the same non-trivial short-stay marketplace two ways: once as raw AI-generated code and once on Buzzy. The point was not to prove that an app can be generated. It was to compare what each model leaves the organisation to maintain.

Benchmark finding	Raw AI-generated code	Buzzy
Creation speed	A working application was generated quickly, then expanded to parity.	The app was assembled from data model, screens and native platform components.
Custom code surface	~5,200 lines of application code; ~6,200 with tests.	Minimal custom extension footprint for the customer to maintain. For portfolio modelling, this paper assumes 5% of the raw-code footprint: approximately 260 lines per app, mainly for backend integrations, bespoke widgets or unique logic not yet covered by native components.
Dependency burden	50+ direct and transitive dependencies, including a day-one high-severity dependency issue requiring a breaking upgrade.	Platform-managed dependency and runtime layer.
Security and compliance surface	Independent audit found one critical vulnerability and seven further issues in the hardened raw build; compliance evidence would also need to be assembled per app.	Built-in security controls and compliance assessment are informed by the app definition, making risks and capabilities easier to inspect.
Data and infrastructure	Database lifecycle, migrations, backups, HA, hosting, TLS, CI/CD, secrets and monitoring must be handled per app.	Platform handles common data, runtime and lifecycle capabilities.
Cost shape at scale	Maintenance grows with the number of generated codebases.	Core improvements and fixes can benefit many apps at once.

The controlled benchmark supports the core TDI argument: raw AI generation can look efficient at the point of creation while increasing structural, operational and organisational debt over the application lifecycle.

9. Portfolio implication: the code surface still scales

The ShortStay benchmark gives a simple unit of comparison. The raw AI-generated Airbnb-style app reached feature parity at approximately 5,200 lines of application code, or approximately 6,200 lines including tests, with 50+ direct and transitive dependencies. In the controlled comparison, Buzzy covered the common application patterns through platform capabilities.

For a more realistic enterprise planning model, this paper assumes Buzzy apps still require some custom extension code: backend integrations, bespoke widgets and unique logic not yet covered by native components. The modelling assumption used here is 5% of the raw-code footprint: approximately 260 lines per app, or approximately 310 lines including tests.

Portfolio punchline
The point is not disappearing code; it is a far smaller custom-code surface. In this model, Buzzy carries roughly 5% of the raw custom-code footprint while common security, compliance assessment, testing, APIs, agentic-flow controls and upgrades sit in platform-managed capabilities.

Similar apps	Raw AI-generated code surface	Buzzy estimated custom extension surface (5%)	Custom code surface avoided
1	~5,200 lines	~260 lines	~4,940 lines (95%)
10	~52,000 lines	~2,600 lines	~49,400 lines (95%)
50	~260,000 lines	~13,000 lines	~247,000 lines (95%)

At 50 similar apps, the illustrative difference is approximately 260,000 lines of raw app code versus approximately 13,000 lines of Buzzy custom extension code. The important point is not the exact line count; it is the shape of the curve. Raw generated code scales the maintenance surface with every app. Buzzy concentrates recurring capability into the platform and keeps app-specific code small, explicit and inspectable.

Buzzy will continue to enhance and commoditise native components where repeated patterns make sense. That means today's recurring custom extension can become tomorrow's platform-maintained capability, further reducing the custom footprint for future applications. The detailed calculator and assumptions are included in Appendix B.

10. Decision framework

The objective is not to ban AI-generated code. The objective is to use the right lifecycle approach for the right work. Fully custom code remains appropriate where the implementation itself is the strategic asset. A platform model is more attractive when the business value sits in the workflow, data, user experience and pace of change rather than in bespoke reimplementation of common application plumbing.

Use case	Recommended default	Reason
Throwaway prototype	AI-generated code or design prototype.	The goal is learning, not long-term operation.
Genuinely novel core technology	Custom engineering.	The implementation details may be the differentiated IP.
Line-of-business workflow app	Buzzy first, with controlled extensions where genuinely needed.	Patterns repeat; common maintenance burden should be centralised while unique logic stays explicit.
Customer-facing app with standard components	Buzzy first, with extension points where needed.	The business needs speed, quality and lifecycle control; any custom code should be limited to the parts that are genuinely unique.
Vendor-delivered application	Buzzy-enabled delivery where feasible.	Keep delivery leverage while reducing bespoke code dependency.
Regulated or sensitive environment	Buzzy with appropriate deployment model and governance review.	Centralised controls reduce inconsistent per-app security implementation.

Questions to ask before approving the next AI-built app

1. What is the Technical Debt Index impact across structural, operational and organisational dimensions?
2. What is the expected Technical Debt Ratio over three years, not just the initial build cost?
3. Who patches the next critical vulnerability, and how many applications must be touched?
4. Which capabilities are genuinely unique, and which are common patterns we should not rebuild?
5. If the vendor or developer changes, what durable application knowledge remains?
6. How will this application be governed when there are ten, fifty or one hundred similar apps?
7. If this becomes a portfolio pattern, what is the five-year TDR under low, medium and high remediation assumptions?

11. Conclusion

AI has made writing software easier. It has not made operating and maintaining software easier. The organisations that confuse those two ideas will create application estates faster than they can govern them.

The strategic response is not to reject AI. It is to adopt an architecture that turns AI speed into durable capability. That means measuring technical debt, reducing per-app code maintenance, centralising common controls, and treating reusable application capabilities as assets rather than rebuilding them for every project.

Buzzy's proposition is therefore not simply "build apps faster". It is: build more applications, maintain less bespoke code, and reduce the Technical Debt Index that would otherwise accumulate as AI-driven application delivery scales across the organisation.

Final positioning

AI should accelerate business capability, not multiply software liability. Buzzy helps organisations say yes to AI-speed delivery without saying yes to unmanaged code sprawl.

Appendix A: Technical debt measurement checklist

This checklist can be used as a supplementary artefact alongside the whitepaper. It helps a buyer score whether a proposed AI-generated, vendor-built or internally built application is likely to increase the organisation's Technical Debt Index.

Dimension	Questions	Example evidence to request
Structural	Is the architecture documented? Are data models explicit? Are dependencies understood? Is logic duplicated? Are common patterns reused?	Architecture diagram; data model; dependency inventory; code quality report; reuse plan.

Dimension	Questions	Example evidence to request
Operational	Are tests defined? Is CI/CD in place? How are secrets, monitoring, backups, compliance checks and incident response handled? What is the patching process?	Built-in test report where available; pipeline configuration; security and compliance assessment; backup/restore evidence; monitoring and runbook.
Organisational	Who understands the system? Is the brief, flow, data model, blueprint and design system captured? What happens when the vendor or developer leaves? How are changes requested and priced?	Application definition; brief; flows; blueprint; data model; theme/design system; support model; change process; vendor exit plan.
TDR / TCO	What does the app cost over three years? What remediation and upgrade work is likely? What assumptions are hidden in the initial build estimate?	Three-year TCO model; remediation estimate; upgrade forecast; support staffing model.

Simple TDI scoring approach

A lightweight score can be enough to change the conversation. Score each dimension from 1 to 5, where 1 means low visible debt and 5 means high or unmanaged debt. The purpose is not false precision. The purpose is to force hidden lifecycle assumptions into the decision before the organisation approves another application.

Score	Meaning	Interpretation
1	Low debt	Common capabilities are reused, governance is clear and lifecycle accountability is explicit.
2	Managed debt	Some bespoke work exists, but risks are understood and resourced.
3	Moderate debt	Lifecycle obligations exist but are only partly planned.
4	High debt	Security, testing, upgrades or handover are weakly defined.
5	Unmanaged debt	The app can be demonstrated, but nobody can clearly explain how it will be governed, tested and maintained.

Appendix B: Portfolio-scale TDR model and assumptions

This appendix turns the ShortStay / Airbnb-style benchmark into a simple portfolio planning model. It is illustrative, not a forecast. Buyers should replace the assumptions with their own build costs, remediation rates, testing standards, integration requirements and support models.

The purpose is to make the hidden scaling effect visible: when every application creates its own codebase, every application also creates its own security, testing, documentation, dependency, release and handover burden. Buzzy reduces that burden by minimising the app-specific custom code surface and shifting common capabilities into platform-maintained components.

Input	Raw AI-generated app	Buzzy modelling assumption
Application code	~5,200 lines per app	~260 custom extension lines per app (5% of raw)
Application code including tests	~6,200 lines per app	~310 lines including tests (5% of raw-with-tests)
Dependency surface	50+ direct and transitive dependencies per app	Common runtime and dependency layer is platform-managed; app-specific extensions may still introduce controlled dependencies
Custom work that remains	Most application behaviour is encoded in the generated codebase	Backend integrations, bespoke widgets, unique logic and exceptional patterns not yet covered by native components
Strategic direction	Each new app adds another codebase	Repeated extension patterns can be commoditised into native components over time

Portfolio calculation summary

B1. Five-year TDR-style debt-service model

For modelling purposes, set one raw app build to 100 effort units. A raw generated application carries a custom-code footprint factor of 1.00. A Buzzy application in this scenario carries a custom extension footprint factor of 0.05. The remediation-rate assumption is the annual effort required for security fixes, dependency patching, testing, documentation, release work, integration upkeep and framework upgrades.

Portfolio remediation cost = number of apps × raw-app build effort × custom-code footprint factor × annual remediation rate × years.

At a 35% annual remediation rate over five years, the raw-code portfolio accumulates 175% of the original build effort again as debt service. Under the 5% Buzzy custom-extension assumption, the app-specific custom-code debt service is 8.75% of the raw app build effort per app.

Scale	Raw code surface	Buzzy custom extension surface	Raw 5-year debt service at 35%/yr	Buzzy 5-year custom-code debt at 5% footprint
1 app	~5,200 lines	~260 lines	175 units	8.75 units
5 apps	~26,000 lines	~1,300 lines	875 units	43.75 units
10 apps	~52,000 lines	~2,600 lines	1,750 units	87.50 units
20 apps	~104,000 lines	~5,200 lines	3,500 units	175.00 units
50 apps	~260,000 lines	~13,000 lines	8,750 units	437.50 units
100 apps	~520,000 lines	~26,000 lines	17,500 units	875.00 units

B2. Sensitivity: low, medium and high remediation assumptions

The exact remediation rate will differ by organisation, risk profile and application complexity. The key observation is that Buzzy's custom-extension assumption changes the slope of the scaling curve. The table below shows raw versus Buzzy custom-code debt-service units under three remediation-rate scenarios.

Annual remediation rate	50 raw apps	50 Buzzy apps @5%	100 raw apps	100 Buzzy apps @5%
15%	3,750 units	187.50 units	7,500 units	375.00 units
35%	8,750 units	437.50 units	17,500 units	875.00 units
60%	15,000 units	750.00 units	30,000 units	1,500.00 units

B3. How to use the calculator

- Replace the 100 effort-unit baseline with the organisation's estimated build cost for a comparable application.
- Replace the 35% remediation rate with the organisation's expected annual support, testing, security, upgrade and release burden.
- Adjust the Buzzy custom-extension footprint above or below 5% based on the amount of backend integration, bespoke widget work and unique business logic required.
- Revisit the assumption over time. As Buzzy turns repeated custom patterns into native components, the custom-extension footprint can fall for future applications.

Appendix takeaway: a 5% custom-code assumption still produces a large portfolio effect. For 50 similar apps, the model moves from approximately 260,000 raw lines to approximately 13,000 Buzzy custom extension lines. The claim is not that custom code disappears. The claim is that the custom-code surface becomes much smaller, more explicit and easier to govern.

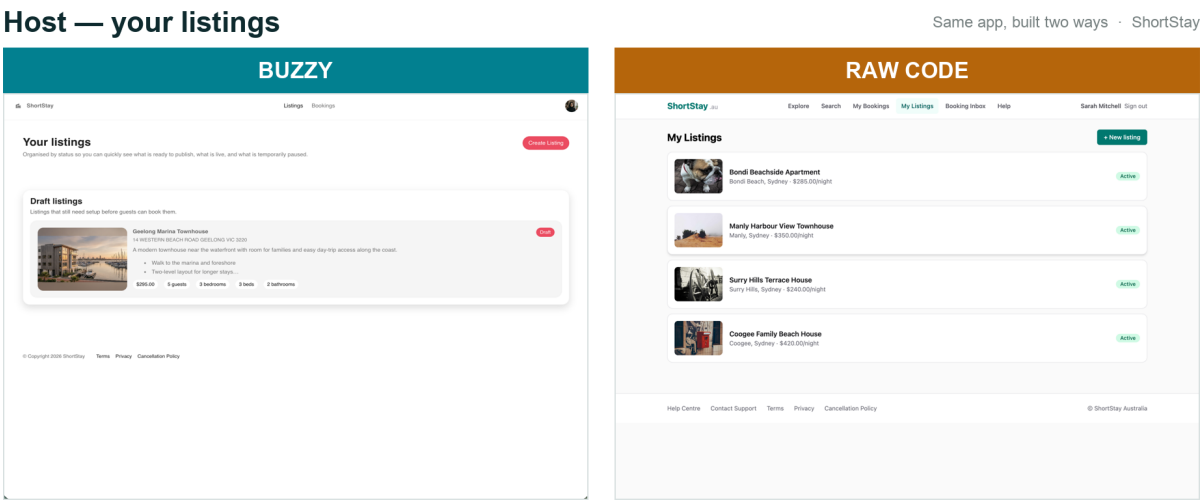
Appendix C: Source notes and external references

This appendix records the external references used in the main paper. The goal is not to turn the whitepaper into an analyst report; it is to add just enough third-party context to show that technical debt measurement is now a real, visible category.

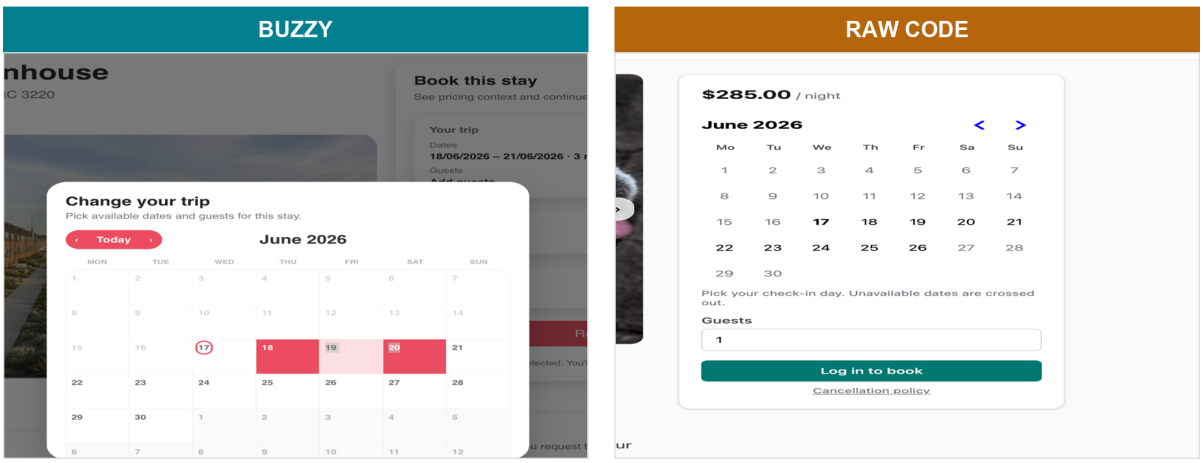
Source	Key statistic or framing used	How it supports the paper
Gartner Magic Quadrant for Technical Debt Management Tools, 20 May 2026	Technical debt management is defined around analysing source code, architecture and dependencies; Gartner projects architectural technical debt will	Validates that technical debt has become a recognised software category and that architecture-level debt is the strategic issue, especially as AI coding expands.

	account for 80% of all technical debt by 2027; Gartner estimates the market will reach $1.2B in annual revenue by the end of 2026, with double-digit annual growth through 2030.	
DX, Technical debt ratio: How to measure technical debt, updated 28 Oct 2025	TDR = (Remediation Cost / Development Cost) x 100. DX suggests below roughly 5-10% generally supports healthy velocity, while above 20% typically indicates systemic issues.	Supports using TDR as an executive signal rather than a purely engineering metric.
vFunction, How to Measure Technical Debt, 11 Jul 2024	vFunction frames technical debt across code, design, testing, documentation and architecture, and highlights complexity, risk and overall debt as useful architectural measures. It also describes a minimal TDR below 5% as ideal.	Supports the paper's structural / operational / organisational debt lens and the need to measure architecture, not just lines of code.

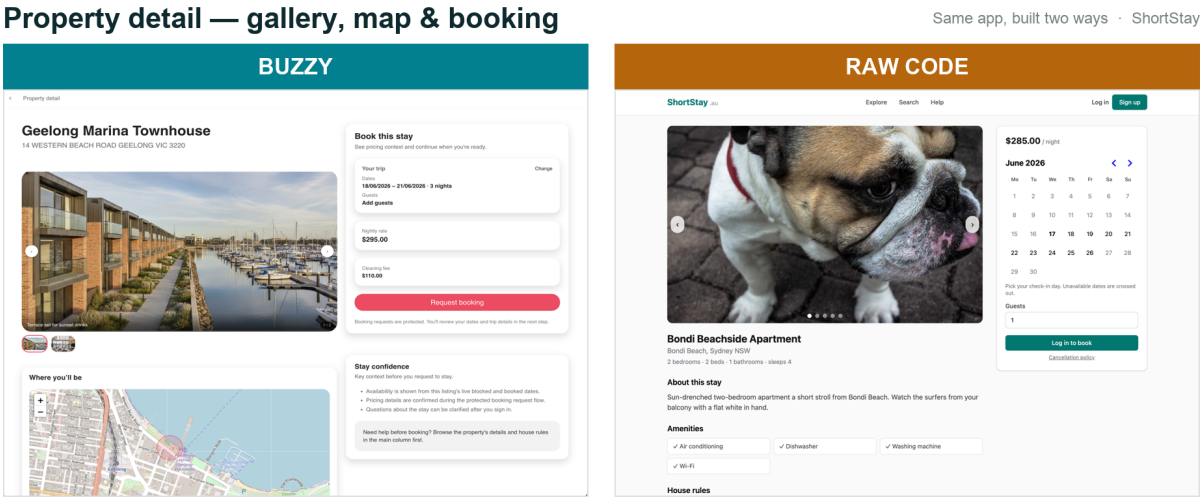
These sources are included to strengthen the reader's confidence that technical debt measurement is an emerging management discipline, not just a vendor narrative. None of these sources should be read as an endorsement of Buzzy.



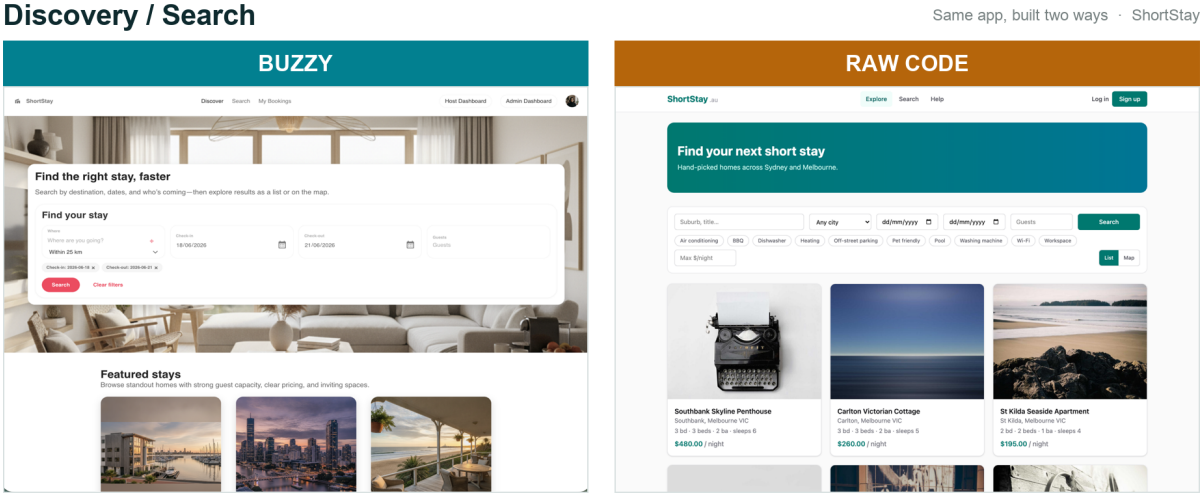
D4. Host — your listings



D3. Booking — availability calendar



D2. Property detail — gallery, map & booking



D1. Discovery / Search

The following screenshots provide additional visual context for the controlled ShortStay benchmark used in this paper. Each example shows the same functional surface built with Buzzy and with a raw-code approach.

Appendix D: ShortStay benchmark screenshots

About Buzzy

Buzzy is an AI-native semantic application platform for building and running web and native applications from a single application definition. You describe the app — its brief, flows, blueprint, data model, theme/design system, screens, permissions, tests and integrations — and Buzzy assembles it from maintained platform capabilities for the patterns business applications repeatedly need: forms, lists, dashboards, search, calendars, scheduling, maps, messaging, security, compliance assessment, data, APIs, agentic flows and lifecycle operations.

Because those capabilities are built once and maintained centrally, the customer can reduce the day-to-day burden of securing, testing, documenting, patching, upgrading and operating bespoke code for each application. Where a project needs backend-specific integrations, bespoke widgets or unique logic, Buzzy supports controlled extension points so the custom code that remains is small, explicit and easier to govern.

The result: organisations can use AI to accelerate application delivery while minimising the bespoke code maintenance burden that usually compounds with every new app. As repeated extension patterns emerge, the

Buzzy team can continue to enhance and commoditise native components so future apps carry even less custom code.

Explore Buzzy: www.buzzy.buzz