

Mosterd — The Model Is the Application

A model-driven platform for business registers — agent-ready by design.

A whitepaper on building serious business applications in the age of AI — and why the answer is a smaller surface, not more code.

Executive summary

AI has made writing software dramatically cheaper. It has not made *owning* software cheaper. The database migrations, the access control, the security patches, the search infrastructure, the hosting, the audit trail — the cost of a business application was never in typing the code, and AI does not remove that cost. It hides it until the first incident.

The Mosterd Platform takes a different position. It is a multi-tenant SaaS engine on which you build a complete, hosted business application **without writing an application at all**. You describe your domain — the objects, fields, relations, business rules, screens — as a compact set of JSON definitions in a Git repository. The platform turns that model into a working product: database, REST API, generated user interface, authentication, document handling, search, exports, and AI-powered data processing.

The result is a system where the part *you* own is small, declarative, reviewable, and versioned — and everything expensive is shared, hardened, and maintained once, for every tenant. And because the entire application is derived from an explicit model, it is also the natural foundation for what comes next: business data that AI agents can access safely, because the platform knows exactly what the data means and who may see it.

1. The 80% every business system shares

Strip any serious business application — a contract register, a CRM, an incident log, a corporate housekeeping database — down to its essentials and you find the same anatomy every time:

- Structured objects with typed fields and relations between them
- Lists you can filter, sort, search, and export
- Forms with validation, detail pages, dashboards
- Files and documents attached to records
- Rules: computed fields, statuses, actions, notifications
- Users, roles, authentication, an audit of who did what
- An API, because no system lives alone anymore

Teams rebuild this anatomy over and over, and the rebuild is never the hard part. The hard part is everything around it: keeping it fast when the dataset grows, keeping it secure as threats evolve, migrating data as the model changes, running it in production for years. That is where budgets go — and where self-built systems quietly decay.

The Mosterd Platform implements this shared anatomy **once**, as a product, and lets you supply only the part that is genuinely yours: the model of your domain.

2. What building on the platform looks like

You produce exactly one artifact: a Git repository containing your data model.

A model is a set of **classes**, defined in JSON. A class declares its fields (text, dates, amounts, currencies, files, rich text — fifteen value types), its relations to other classes, computed fields via formulas and templates, list views, page layouts, and scripts that run on create, update, delete, or at the click of a button. A built-in `commons` library provides ready-made building blocks — persons, companies, documents, invoices, emails — that your classes extend.

A fragment of a contract class, to make that concrete:

```
{
  "class": "legal.contract",
  "type": "BASIC",
  "inherits": "commons.document",

  "valueTypes": [
    { "name": "contractNumber", "required": true },
    { "name": "contractValue", "dataType": "CURRENCY" },
    { "name": "endDate", "dataType": "DATE" },
    { "name": "isExpired", "dataType": "BOOLEAN" }
  ],

  "formulas": [
    { "valueType": "isExpired", "formula": "inPast(endDate)" },
    { "valueType": "name", "template": "Contract {{contractNumber}} – ends {{endDate}}" }
  ],

  "relations": [
    { "name": "counterparty", "type": "commons.company", "required": true }
  ],

  "scripts": [
    { "name": "notifyExpiry", "type": "consumerGateway", "gateway": "Mail",
      "trigger": "UPDATE", "preconditions": ["isExpired"] }
  ]
}
```

Do not mistake the brevity for simplicity — every line above is leverage. From this fragment alone, the platform derives the database schema, the API endpoints, the search index, the list views, and the forms. `isExpired` is not a stored flag someone has to maintain: the engine knows the exact day its value will flip and re-evaluates it precisely then — no scheduler, no cron job. The display name is regenerated whenever its underlying values change; the platform maintains a dependency graph of which values affect which objects, and re-evaluates only what is affected. And the mail script fires automatically the moment a contract expires. A good developer with a

modern framework builds the simple version of this in a day — that is not where the leverage is. The leverage is in what does not fit in that day: the self-scheduling re-evaluation, the cross-object dependency tracking, the search index, the API, the generated screens — and in the fact that the fortieth field, rule, and screen costs as little as the first.

Push a commit, and the platform reloads the model via a webhook. There is no build step, no deployment pipeline to maintain, no framework upgrade ever again. From your model, the platform derives:

- **A REST API** with rich querying, filtering, aggregation, full-text search, and Excel export
- **A complete web application** — tables, forms, dashboards, document viewers — rendered dynamically from your layout definitions
- **Business logic** — JavaScript scripts with a clean API, triggered by events or users
- **AI processing** — built-in gateways that extract structured data from documents, classify and triage files, parse invoices, and enrich records from official company registers; multi-step AI pipelines are declared directly on a class, with token usage and cost tracked automatically per call

Because the model lives in Git, it behaves like code without being a codebase. Branches give you development, staging, and production environments. One repository serves many customers. Optional **modules** let tenants switch feature bundles on or off — one product, many configurations, a single source of truth.

3. "But with AI, we can just build our own"

This is the objection we hear most, usually from good engineers. It deserves a serious answer. There are four.

First: AI lowers the cost of the wrong thing. AI is superb at producing the first working version — the part that was already the cheapest. What it does not produce is five years of operations: dependency patching, schema migrations under live data, performance tuning at scale, penetration testing, incident response, backup discipline, upgrade paths. A generated codebase is still a codebase, and from day one it is *your* codebase — including the parts no one on your team has actually read.

Second: a platform gives AI a better target. Ironically, the strongest argument for the platform is AI. Asking an AI to generate and maintain a full application means trusting tens of thousands of lines it wrote at speed. Asking an AI to write a Mosterd data model means asking it for a few hundred lines of declarative JSON against a documented, validated schema — small enough to review line by line, constrained enough that there is little room to go wrong, and versioned in Git so every change is a diff. The platform's documentation is explicitly written to serve as AI context. You still build with AI. You just point it at your domain instead of at plumbing.

Third: the expensive properties are amortised. Security monitoring, penetration tests, hardened hosting procedures, performance engineering — the platform model allows each of these to be invested in once and inherited by every tenant. A self-built system must fund all of it alone, forever, or go without. Most go without.

Fourth: it is proven, not promised. The platform is not a code generator producing untested output. It is one engine — a fully reactive backend on a dual SQL/NoSQL persistence architecture — exercised across every tenant, so every improvement is battle-tested before it reaches you. The applications we have migrated onto it run markedly faster than the conventionally built systems they replaced, including on large datasets. That is a claim we would rather demonstrate than assert: bring a dataset of your own size and see for yourself.

Where AI app builders and backend platforms fit

The question usually arrives with names attached, so let us be precise rather than defensive. Two categories deserve credit for what they do well.

AI app builders (Lovable and its peers) turn a prompt into a working full-stack application, complete with front end, database and login. They are genuinely impressive, and on time-to-first-screen they beat us — we would not claim otherwise. But look at what you hold afterwards: a generated codebase. The AI wrote it quickly; it is nonetheless yours to patch, migrate, secure and understand for as long as the business depends on it. Their output is a starting point that begins ageing immediately. Ours is a model that stays the source of truth — the application is derived from it at runtime, so it never drifts out of sync with the description, and a change to the description is a change to the application.

Backend platforms (Supabase, Firebase and similar) give you a managed database, authentication, storage and generated APIs — excellent infrastructure, and a sensible foundation for a team that intends to build. The distinction is what they deliberately leave to you: the application. The screens, the business rules, the workflows, the reporting — everything above the database is your project, and your maintenance. We do not compete with that layer so much as sit above it: what they call your job is what we generate.

The clearest single contrast is authorization. A backend platform gives you row-level security — SQL policies, evaluated per query, written by someone fluent in SQL. We let you declare access in the same terms as your business ("this role works with these dossiers and everything beneath them"), materialised when data is written so reads stay cheap. Theirs is more powerful in the hands of an engineer; ours is comprehensible to the person who actually knows who should see what.

So the honest summary: if your goal is to build software and you have the team to own it, those tools are good and you should use them. If your goal is to *have* a business register that stays correct, current and yours for a decade, owning less of it is the point.

4. The 90% wall

Experienced developers carry a second, well-earned objection: *"Generic platforms get you to 90% fast. Then come the specific business rules and flows, you hit the platform's ceiling, and the application never gets as good as a custom one."* For most no-code and low-code tools, that is exactly how it ends.

The platform's answer is that its escape hatch is not a hatch — it is a full business tier. JavaScript can be injected at the decisive points of the data flow: on create, on update, on delete, as user-triggered actions, as data sources, with preconditions, parameters, and access to a clean API for querying, creating, and updating any object, generating files, and calling external services. Where a typical platform offers a constrained expression language and a dead end, this platform offers what you would otherwise write in the service layer of a custom application — without adopting the codebase around it. The direction of travel extends this further: scripts that validate *in advance* whether an action is permitted at all, closing the last gap with a hand-built business tier.

The same philosophy applies to the user interface. The generated application covers the standard screens, but those screens are themselves built from a published library of portable web components — input controls, dynamic forms, interactive tables with filtering and virtual scrolling, complete dashboard and document-view elements — that work in any modern stack. If your product needs its own front end, you build your own app

against the platform's API, using the same components the platform's UI is made of: custom where it differentiates you, generated where it doesn't.

Can a model be tested? The fair follow-up, since a declarative model is still a thing that can be wrong. Because a tenant costs almost nothing to create, a model can be verified against a disposable environment rather than a person's attention: check that every reference resolves, that every label exists in every declared language — a missing translation is the defect that never fails loudly, it just renders a plausible word — and that the flows you care about actually work. The first phase makes a change verifiable by whoever made it, before it reaches anyone else (*in development*); the same machinery later serves builders who have no local installation at all. "Define a model, get an application" is a weaker promise if finding out whether it works means installing four databases first.

The honest evidence, with its honest caveat: the applications running on the platform today — contract management, corporate housekeeping, entity registers, service desks — have not hit a rule or flow that could not be expressed. Those applications were built close to the platform, so read that as a strong signal rather than proof. But it is a testable signal: the failure mode of generic platforms shows up early, so bring your hardest business rule to a pilot and try to break it.

5. Compute at write time, read instantly

Business data is read far more often than it is written. The platform turns that asymmetry into an architectural principle: spend the computing power at the moment of writing, so every subsequent read is instant. This principle produces a set of capabilities that are notoriously time-consuming to build yourself:

- **A formula and template engine that materialises results.** Complex calculations and even complete generated texts (via Handlebars templates) are computed when data changes and stored on the object — deliberately redundant, so lists, searches, and exports never pay the price at read time.
- **Time-aware formulas that keep themselves current.** A formula like `inPast(expiryDate)` returns `false` today — and the engine registers the exact moment that answer will change, then re-evaluates precisely then. Statuses flip on the right day without cron jobs, polling, or a developer remembering to build a scheduler.
- **A dependency graph across objects.** The platform records which objects are affected by a change to any value or relation, and re-evaluates exactly the affected formulas — efficient, targeted, and automatic, even across relations.
- **Rule-based publication to external stores** (*available today via the replication gateway*). A derived subset of the data can be continuously replicated to an analytical database such as Google BigQuery, giving dashboard builders near-live data without ever touching the operational system.
- **Hierarchical relations that answer impossible queries.** Relations can traverse recursive parent chains (`inherit`, optionally in both directions): filter on *Europe* and find every company linked to any territory beneath it — the kind of query that in a conventional relational schema means recursive SQL, custom indexing, and a week of work, and here is a single model property.

6. Security and data ownership as architecture

For a decision maker, the security question is not "is it secure today" but "who keeps it secure, at whose cost". The platform's answer: multi-tenant isolation at every layer, OAuth2 authentication throughout, sanitisation of all user-supplied content, rate-limited external calls — maintained centrally, audited centrally.

Access control derived from your model, not bolted onto it. Permissions are declared in the model as rules over the relationships in your own data — "this role works with these dossiers and everything beneath them", "a case handler sees the files of their own clients" — and the platform materialises them when data is written, so a read is a single cheap set intersection rather than a permission calculation per row.

This is in production and enforcing. Access spreads along the relations you declared, however many hops away; rules can be conditional on the record's own state, and a condition is checked against the state a write would *produce*, not the one it started from. Reads and searches show only what the caller may see; creating, changing and re-parenting are authorised against the permissions the result would carry; deleting a record takes down the access it handed to everything beneath it. The same decision the server enforces is published to the interface, so a button is never offered that the server will refuse — a capability published from a different rule than the one enforced is precisely how a UI ends up promising what the backend denies. And because access is materialised rather than inferred, the platform can answer the question auditors actually ask: not just *who* may see this record, but *along which path* they were granted it.

One boundary is drawn deliberately, and we would rather state it than let you discover it. The filter decides which records you may *open*; it does not filter every fact a record volunteers about what it links to — a related record's display name travels with the record that links to it. Closing that would mean an access check per link per row, hundreds of them to serve one page, on a read path built to be a single intersection. So it is documented as a modelling rule instead: a neighbour whose very existence is confidential must not be directly related to a record the reader can open. That is a design constraint we hand to whoever builds the model, not a defect waiting on a release.

That an authorization model of this depth can be *declared* rather than implemented — and then arrive complete in every application on the platform at once — is the platform argument in miniature.

Two further capabilities on the roadmap push this beyond what self-built systems realistically reach:

- **Exfiltration resistance** (*direction*). The classic risk of any data platform is that a compromised account quietly drains an entire database. Planned countermeasures treat bulk access as a first-class threat: comprehensive access auditing and circuit-breaker patterns that detect and halt anomalous extraction.
- **Documents your provider cannot read** (*direction*). The platform's binary storage service is designed to be deployable **outside the platform's own domain** — including on a customer's own infrastructure, with its own backups, and with downloads restricted to the customer's network. The platform then never touches document contents; it only issues access keys. The customer remains the owner of their data — not contractually, but architecturally.

7. Where this is going: a register AI agents can trust

The platform's strategic direction is not to make itself maximally clever. It is to make your data maximally *usable* — including by AI agents acting on your behalf.

Today, connecting an AI assistant to business data means hand-building integrations that the AI has to guess its way around. The platform is in a fundamentally better position: it holds an explicit, machine-readable model of what every object means, how objects relate, which operations are valid, and — with relation-based access control — who may see what.

The **model-driven agent interface** (*direction*) builds on exactly that: any tenant's model can be published as a set of tools an AI agent understands — not just fields and types, but written semantics: what this domain is, which operations make sense, which invariants must never be violated. Combined with per-user access scoping and a full audit trail, this turns every application built on the platform into something rare: **a system of record that an AI agent can query and update safely** — from tools your team already uses, without a bespoke integration project.

Two design choices follow from that (*direction*). First, the register **publishes; it does not insist on its own agent**. The tool surface is exposed through the open Model Context Protocol, so whichever assistant or agent platform your organisation standardises on can use it — and switching assistants does not mean rebuilding the integration. Second, **agents get less than people, not more**: access is delegated on behalf of a named user, for a bounded time, never beyond that user's own rights, and any operation can be set to require human approval before it takes effect. Because those rules are enforced in the data layer rather than in the agent, they hold no matter which agent is asking.

You cannot retrofit this onto a hand-built codebase without rebuilding half of it. On a model-driven platform, it is the natural next step.

8. Straight answers to fair questions

Anyone evaluating a platform should ask hard questions. Here are ours, answered without spin.

"Aren't we locked in?" The model runs on this platform — we won't pretend otherwise. What limits the risk: your data is always exportable in full through the REST API and Excel exports; your model is plain, documented JSON in *your* Git repository, readable by any engineer (or AI) and re-implementable against another stack; and documents can live in a storage domain you control (*direction*), where the platform only ever holds keys, never content. Above all, it is a founding principle of this product that we will not hold data hostage — expect the exit tooling to grow over time, not shrink.

"What happens when the platform evolves — does my model break?" The compatibility burden is ours, not yours, and as of September 2026 that is a mechanism rather than an intention. Every definition file states the format version it was written for, and the platform converts older files on every load — so a model written today keeps loading on later releases without anyone touching it. When we do need to change the format, the platform hands you the whole repository rewritten in the new one, with a report of what changed; you review a diff instead of running a migration. A model that is invalid after conversion is refused with every error listed at once, file and location included, and the previously loaded model keeps serving your application — a bad commit means nothing changed, never a broken system. Features are not removed abruptly either: at least twelve months of

warnings first, and after that they are ignored rather than fatal. Branch-based environments let you validate each change in development, test and acceptance before it reaches production.

"What does it cost?" This paper is not a price list, but two things are worth knowing before you compare. First the yardstick: merely *hosting* a self-built application on a major cloud provider quickly runs into hundreds of euros per month — before the first hour of maintenance, patching or monitoring is paid for, and the bill climbs with your dataset rather than with your usefulness. The platform is priced to compare comfortably against that hosting line alone.

Second, and more easily overlooked: **how many meters are running.** The generated-app and backend-platform route bills you on several at once — a subscription, credits consumed per change, database and compute tiers, a hosting provider — each modest on its own and none of them your total. Two consequences follow. Changes cost money *per change*, indefinitely, including the ones spent correcting what the AI got wrong; and per-project pricing multiplies the moment you want a development, test and production environment. Our pricing is one meter, all in, and deliberately dull: the point of a business register is that its cost is something you can put in a budget.

"How mature is this?" The engine runs real applications today; several capabilities in this paper are explicitly marked (*direction*) and are at the design stage. We would rather show than tell: the fastest way to judge maturity is a pilot on your own case.

9. What this means for your decision

Building on the Mosterd Platform is not a trade-off against having "your own" system. The model in your Git repository *is* your own system — reviewable, versioned, portable, and small enough to actually understand. What you give up is ownership of the plumbing: the part that was never a differentiator and always a liability.

	Self-built (with AI)	On the platform
What you own	Full codebase, mostly generated	Declarative domain model in Git
Time to first working version	Weeks—months	Days
Security, pentests, monitoring	Your budget, every year	Amortised across the platform
Performance at scale	Discovered in production	Engineered once, proven across tenants
Schema changes	Migration scripts, downtime risk	Model commit + reload
Specific business rules	Unlimited — you write everything	Unlimited — JavaScript business tier
Temporal logic, derived data	Cron jobs, cache invalidation bugs	Write-time engine, self-scheduling formulas
Live dashboards	ETL project	Rule-based replication to BigQuery
AI agent access	Bespoke integration project	Derived from the model (<i>direction</i>)
Data ownership	On your infrastructure — with your effort	Architectural, incl. customer-hosted documents (<i>direction</i>)

The right question is not "*could we build this ourselves?*" — with AI, you could. The question is "*which part of it deserves to be ours?*" Your domain knowledge does. The other 80% is our product.

© 2026 — *working draft for review. All product claims to be verified against the current platform release before publication.*