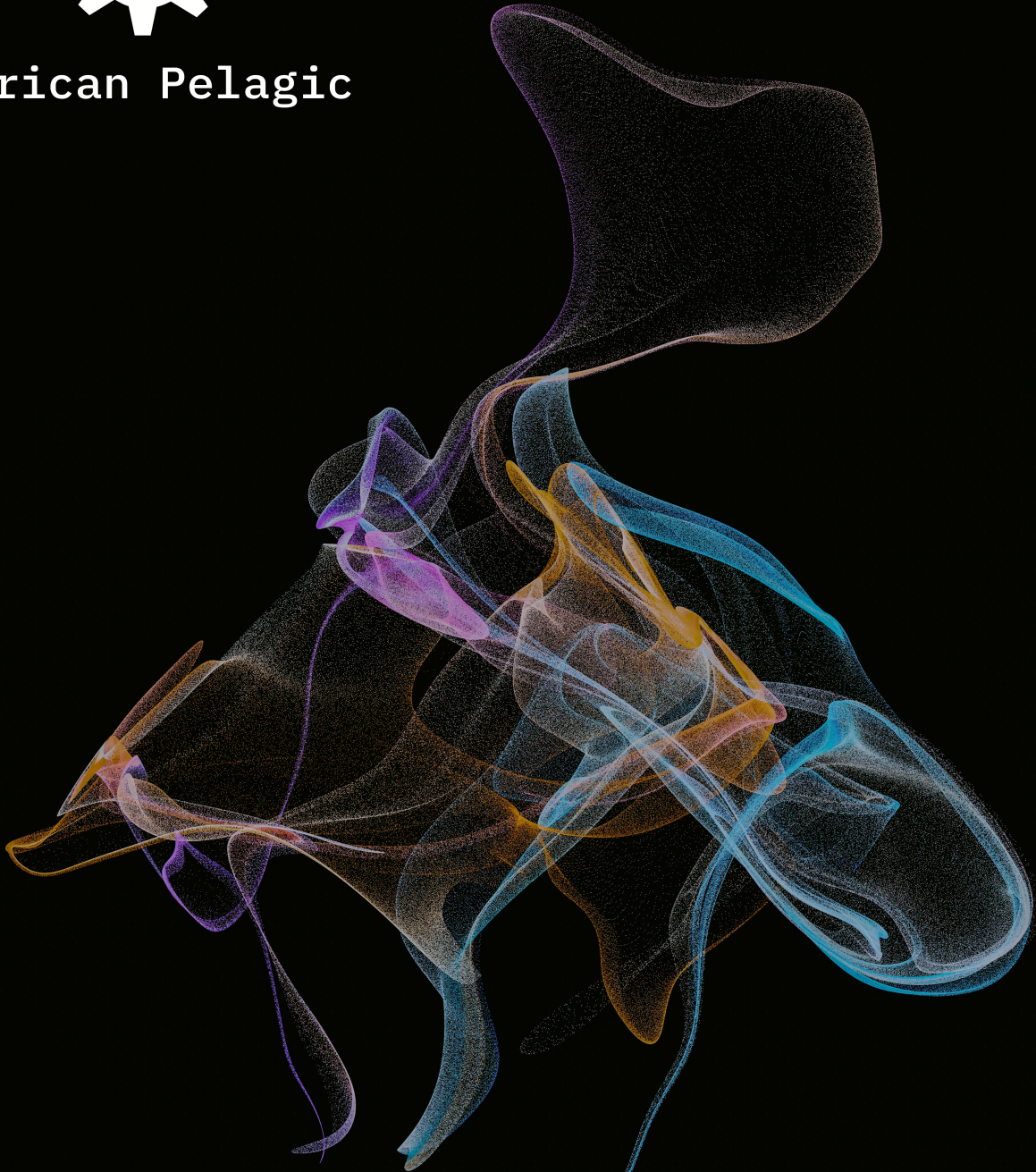




African Pelagic



Machine Learning Company

An Observational Case Study



Contents

1	Executive Summary	2
2	Introduction	3
3	Problem	5
4	Solution	11
5	Outlook	19
6	Conclusion	21



Executive Summary

A mid-sized machine learning company struggled to scale its technology base due to architectural and business-model constraints established during the startup phase. These constraints manifested as declining development velocity, limited innovation capacity, and high deployment and upgrade costs.

Our analysis showed that the company was caught in a self-reinforcing socio-technical loop: early architectural coupling increased coordination costs; those costs shaped hiring, process, and deployment decisions; and those decisions further entrenched the original technical constraints - making resolution increasingly difficult over time.

From a technical perspective, tightly coupled domain models and persistence at the HTTP boundary made local change impossible, ensuring that development velocity would continue to decline unless explicit architectural boundaries were introduced to restore locality of change. Similarly, embedding the ML (machine learning) training pipeline inside the web application constrained experimentation and productionisation, limiting innovation unless orchestration and observability were separated into a dedicated control plane.

Finally, treating bare metal as a first-class deployment target imposed structural limits on cost reduction and operational efficiency. These costs could not be addressed through process improvement alone and would only fall if containerised deployment became the default, with bare metal treated as a premium exception.

Rather than presenting these issues as isolated technical problems, this case frames them as structural dynamics. Continued growth under the existing architecture would compound bottlenecks and fragility. Altering the trajectory required redefining architectural and operational boundaries so that coordination costs could stabilise and efficiency gains begin to accumulate.

This case study has been anonymised to protect confidentiality.



Introduction

This case study examines the company as a socio-technical system, in which architectural design, organisational structure, hiring patterns, and deployment strategy interacted. Rather than treating the symptoms independently, the analysis focuses on the feedback loops connecting technical coupling, coordination cost, and strategic constraint.

The company was growing, however the architectural decisions that had enabled rapid early delivery were beginning to constrain further scale. Development velocity slowed, innovation capacity narrowed, and operational costs increased.

Company Background

The company was a mid-sized machine learning firm providing a document intelligence product that extracted structured data from large volumes of textual records. It had ambitious growth targets that required rapid scaling of its technical foundation. The central challenge was to evolve the existing technology and development practices to support faster change and greater scale, without being constrained by architectural and process choices made for early-stage speed.

Organisational Structure

The company's structure included a management team of mixed technical ability; a research group of data scientists and machine learning engineers; a development team of mainly web developers experienced primarily with the backend framework adopted during the startup phase; and project management, sales, and business development functions.

System Architecture

The company's existing solution consisted of a Django web application, a React-based frontend, and an in-house machine learning pipeline embedded in the web application, deployed on bare-metal servers or containerised on-premises.

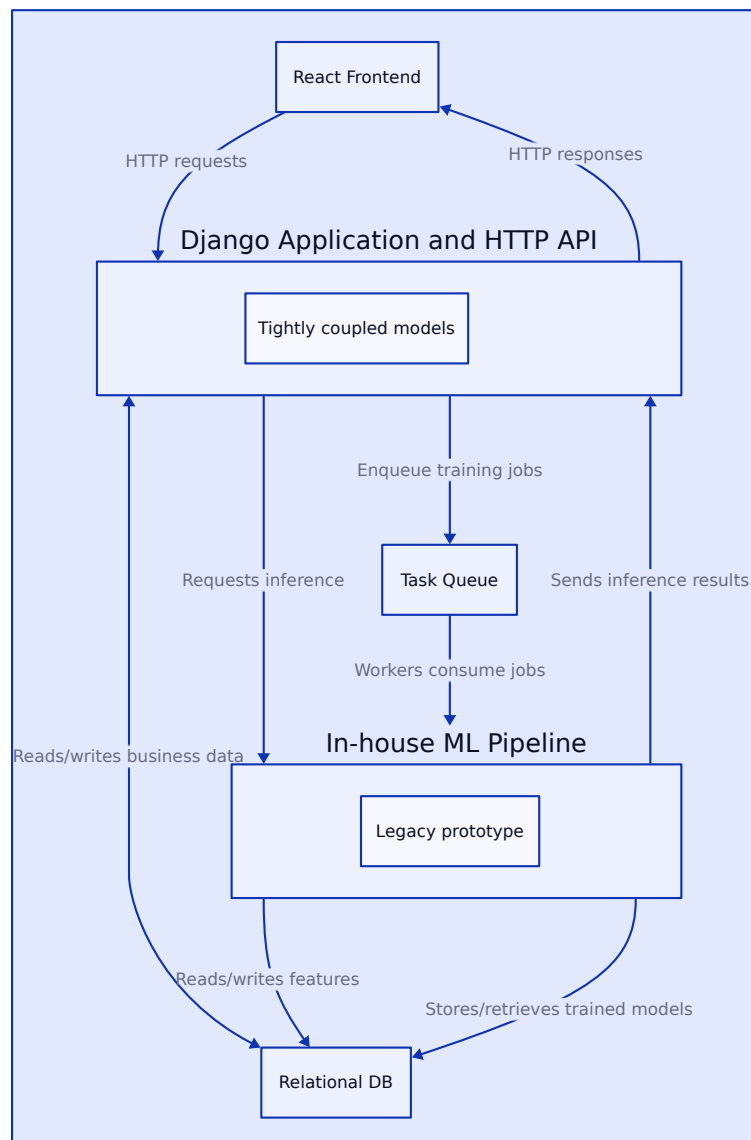
At the time of assessment, the combination of:

- CRUD-oriented modelling and automatic serialization,
- tightly coupled persistence relationships across subsystems, and
- ML pipeline embedded inside the web application



suggested a predictable trajectory: declining development velocity as change cascaded through shared models, rising coordination costs between teams, and constrained innovation due to fragile, environment-sensitive deployments, long restart/test cycles, and a brittle ML algorithm.

Architecture Outline





Problem

Slow Development Velocity

The company's architecture and development practices were producing a clear symptom: feature development was slowing, while management required higher velocity to maintain and grow market position.

Engineers were finding it increasingly difficult to develop new features without modifying existing code; those changes cascaded through the system, causing teams to collide on the same code paths and creating an escalating coordination burden.

This pattern is a well-known enterprise anti-pattern: the Big Ball of Mud. The mechanism is insufficiently rich domain modelling combined with a simplistic persistence strategy, producing strong coupling between subsystems, turning local change into system-wide change.

In this case, development teams had largely followed standard conventions for the chosen back-end framework - Django and its associated libraries - without creating architectural boundaries to compensate for the complexity of the domain.

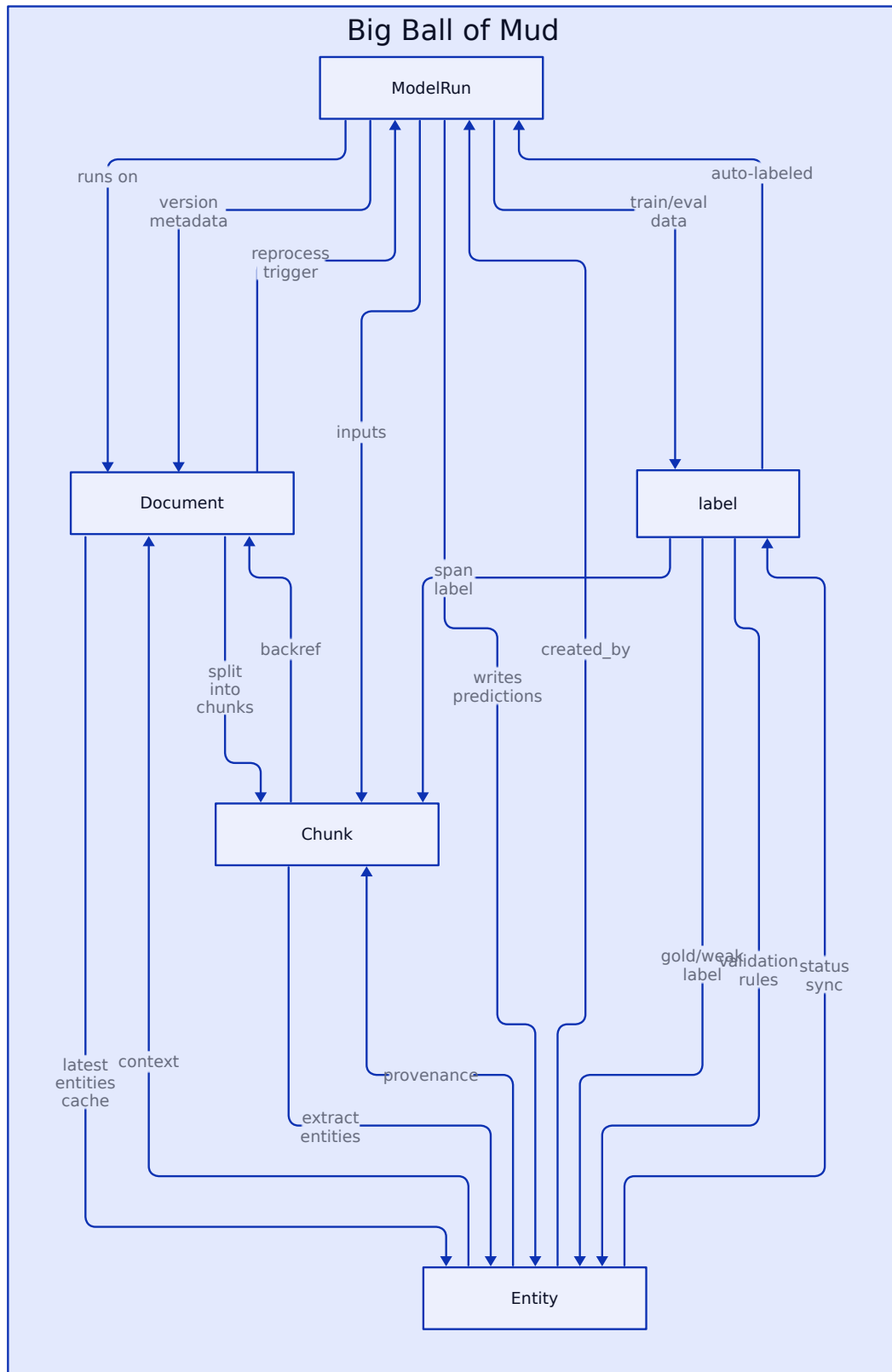
Django is a popular web framework designed for content-driven web applications. It uses a CRUD-oriented Object Relational Mapper that directly maps models to underlying database tables. While this enables rapid early development, its default modelling and persistence strategy does not naturally support complex business domains. Beyond a certain complexity threshold, the standard approach produces ongoing coordination challenges, and the cost of change tends to outweigh the initial productivity benefits.

The Big Ball of Mud

Uncritical use of a CRUD-oriented system like the Django ORM often leads developers to represent a core business concept as a single, globally shared model. As additional subsystems require access to the same data, direct database relationships accumulate, forming a dense web of dependencies. Eventually, even small changes trigger interactions across much of the system, slowing development, complicating testing, and increasing coordination costs.



African Pelagic





Django's ActiveRecord-style persistence exacerbates this by allowing each model to manage its own persistence operations. In complex systems, overlapping modifications across execution flows make application state harder to reason about. Combined with tight coupling, this produces cascading changes and declining development velocity.

Simplistic Serialization Strategy

The company also used Django REST Framework (DRF) for similar reasons of early convenience. Used out of the box, DRF automatically constructs and persists Django models while traversing relationships implicitly. This automation is convenient early on but encourages implicit coupling and further accelerates Big Ball of Mud dynamics.

While suitable for simple data architectures, adapting a system built around default Django and DRF patterns to complex domains often costs more than the early gains justify.

Framework Hiring Bias

Because the system was built using Django and DRF, hiring focused primarily on developers experienced with those tools. Many lacked experience with complex domain modelling and the failure modes of large systems, further delaying recognition and resolution of architectural issues.

Insufficient Innovation

Management also perceived insufficient innovation to maintain technological leadership in the market. Analysis revealed multiple reinforcing factors: research staff were frequently deployed to client sites; the ML pipeline was a fragile in-house system developed during startup; and MLOps was managed inside the Django application via a worker queue, making experimentation-to-production transitions slow and risky.

Training Data Trade-offs

One of the company's key differentiators was its low-data training approach. Models could be trained using as few as 5 to 50 labelled documents, with clients highlighting text spans directly in the user interface. Additional training examples were then inferred from the remaining corpus using a heuristic matching algorithm. While this approach significantly reduced the upfront data burden for customers, it did not generalise reliably without manual adjustment. As a result, each deployment required customer-specific configuration, and clients needed guidance in effective labelling practices.



This product choice had structural implications. By optimising for minimal customer training effort, the company implicitly accepted increased deployment and configuration complexity. Because most customers required on-premises installations, the research team was frequently deployed to client sites to provide training and tune model behaviour. Time that might otherwise have been spent on platform-level innovation was instead consumed by environment-specific configuration and support.

Configuration adjustments were further complicated by the environment-sensitive nature of the machine learning pipeline and the limited ergonomics of its control architecture. Deployments often surfaced unforeseen integration issues, requiring coordinated intervention from both research and engineering teams. Each modification triggered restart and test cycles, slowing iteration and reinforcing the operational load on the team. Over time, this dynamic constrained innovation capacity not through lack of technical capability, but through the cumulative diversion of attention toward customer-specific adaptation.

Lack of MLOps Control Architecture

The second structural constraint on innovation was the absence of a dedicated machine learning operations control architecture. The ML pipeline was managed directly by the Django application, with orchestration logic embedded inside the web layer. As a result, core operational concerns - observability, model management, workflow orchestration, and dynamic resource control - were tightly coupled to application configuration.

Observability was constrained at multiple levels. Pipeline configuration could not be inspected without backend developer access, limiting the research team's ability to understand the precise conditions of a client deployment. During model training, internal state was not visible until completion or failure, preventing iterative diagnosis during execution. Infrastructure resource usage was similarly opaque, making fine-grained or real-time resource adjustment impractical.

Model management capabilities were minimal. Trained models were represented as Django models, with binaries stored in the relational database. Versioning was implemented implicitly through model updates, and previous model artifacts were removed when superseded. This made historical comparison, rollback, and issue diagnosis difficult. Although evaluation metrics were generated during model evaluation and surfaced to end users, there was no dedicated interface for research-led comparison or structured experiment tracking.

Workflow orchestration and resource allocation were implemented using Celery, a worker-queue system configured within the Django application. Pipeline structure and resource parameters were defined statically in application settings, requiring restart cycles to modify execution behaviour. Resource scaling was bounded by worker configuration and the specifications of the underlying bare-



metal infrastructure. Adjustments could not be made dynamically in response to load, performance degradation, or experimental requirements.

Taken together, these characteristics meant that experimentation, deployment, and scaling were mediated through the web application layer rather than a dedicated control plane. This coupling limited visibility, slowed iteration, and constrained the research team's ability to innovate safely and efficiently across customer environments.

Legacy ML Pipeline

The final structural constraint was the legacy in-house machine learning pipeline itself. Developed rapidly during the company's startup phase, the pipeline had not undergone a systematic refactor or strategic reassessment as the organisation matured. Design decisions made for early delivery speed had not been revisited in light of changing business context.

Over time, the pipeline codebase accumulated significant technical debt. It relied on heterogeneous tooling, including multiple package managers, offered limited developer ergonomics, and was difficult to configure reliably. Because it was not containerised, and because many of its dependencies included C and C++ libraries with platform-specific compilation artifacts, the pipeline was highly sensitive to its execution environment. This sensitivity introduced friction into deployment, debugging, and experimentation.

The absence of containerisation and environment isolation limited reproducibility and prevented horizontal scaling. Configuration drift between deployments was common, and modifications often required careful manual intervention. As a result, changes to the pipeline were perceived as risky, particularly given its centrality to the company's value proposition.

The cumulative effect was that the organisation became increasingly reluctant to evolve the very system that defined its core product. Innovation was constrained not by lack of ambition or capability, but by the structural cost and perceived risk of modifying a fragile, business-critical component of the product.

Installation and Upgrade Costs

Installation and upgrade costs were high for both the company and its customers, primarily due to the requirement for on-premises deployment on bare-metal servers driven by security and compliance policies. While this satisfied governance constraints, it limited standardisation and operational leverage.



Without containerisation or cloud infrastructure, deployments became environment-specific and upgrades carried elevated risk due to limited rollback and reproducibility. The resulting complexity increased financial and human resource costs on both sides.

As long as bare metal remained a first-class deployment model, these costs were structurally embedded.

The Cost of Bare Metal

Many of the company's clients required on-premises bare-metal installations. While bare metal can offer performance benefits through direct hardware access, it introduces operational complexity and reduces deployment standardisation.

Machine learning workloads often depend on low-level libraries with platform-specific compilation artifacts. On bare metal, hardware and operating system variation increased the likelihood of compatibility issues, requiring diagnosis, recompilation, and environment-specific adjustments. Installations were rarely uniform, and configuration tended to rely on bespoke automation scripts, which required ongoing maintenance and introduced additional operational risk.

Unlike containerised environments, bare-metal deployments lacked built-in isolation, reproducibility, and rollback mechanisms. Failover configurations required dedicated hardware, horizontal scaling was constrained, and monitoring was less consistent. These limitations increased the risk and effort associated with both installation and change.

Upgrades amplified these challenges. Each update carried elevated risk to data integrity and service continuity, often requiring careful manual coordination. Over time, this operational burden translated into significant time and resource costs for the company, increased consulting overhead for customers, and reduced overall satisfaction with the deployment experience.



Solution

Resolving Slow Development Velocity

Addressing declining development velocity required two simultaneous shifts:

- Prevent further unplanned coupling between domain models.
- Gradually disentangle the existing network of persistence relationships.

Achieving both while maintaining feature delivery and serving existing customers required a strategy that reduced risk rather than amplifying it. Four broad paths were possible:

1. Maintain the status quo.
2. Remove Django and DRF entirely.
3. Introduce a different ORM within Django.
4. Retain Django while restructuring architectural boundaries.

Maintain the Status Quo

Maintaining the existing architecture avoided immediate disruption or resource diversion. However, given the company's growth objectives and increasing domain complexity, this path would have resulted in progressively rising coordination costs and declining development velocity. Resource requirements would have escalated without resolving the underlying structural constraint.

Remove Django and DRF

Replacing Django and DRF with a more flexible framework and persistence strategy would have addressed the root architectural issue. However, the cost of doing so would have been prohibitive.

The codebase relied heavily on tightly integrated framework components, meaning a significant amount of functionality would require reimplementation. The development team's skill profile was aligned with the existing stack, increasing retraining overhead. Furthermore, the data model itself reflected CRUD-oriented assumptions, meaning persistence restructuring would be required alongside framework replacement.

Executing such a transition while maintaining live customer commitments and pursuing growth would have introduced substantial operational risk.



Introduce a Different ORM within Django

Retaining Django while replacing its ORM offered theoretical flexibility. However, Django's ecosystem - including authentication, administration tooling, and DRF - depends deeply on its ORM. Replacing the ORM would effectively invalidate large portions of the existing integration surface.

Because persistence strategy underpins many conceptual and functional aspects of a web framework, altering it in isolation would approach the cost and disruption of a full framework replacement. The economic benefit relative to risk was limited.

Retain Django and Restructure Architectural Boundaries

The most viable path was to retain Django while restructuring architectural boundaries to restore locality of change.

Resolving the "big ball of mud" required reintroducing explicit domain boundaries. Concepts from Domain Driven Design could be applied - specifically Bounded Contexts, Aggregates, and Repositories - to redefine how data was structured and accessed.

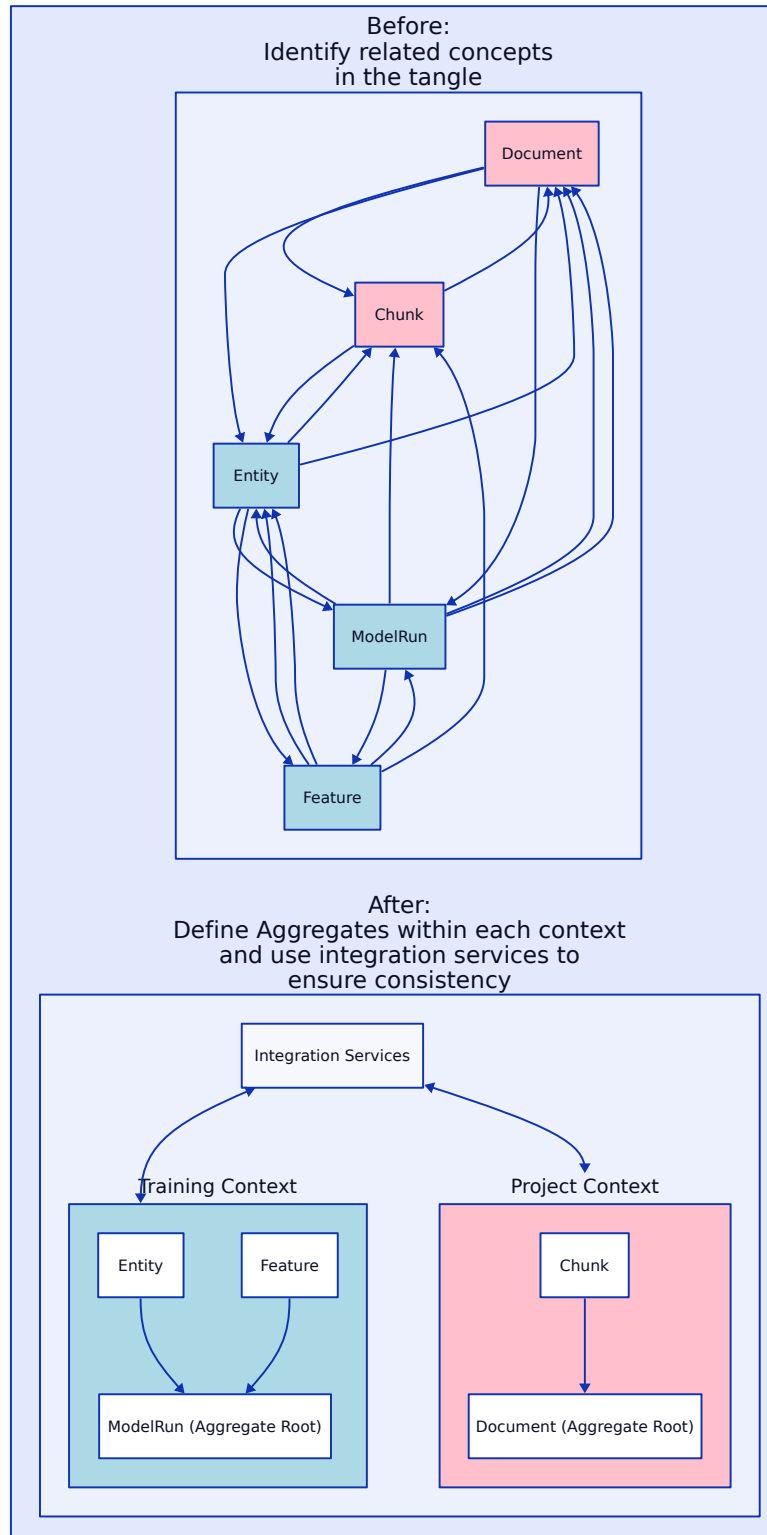
Bounded Contexts establish clear conceptual zones within a system based on conceptual adjacency. Aggregates define which entities changed together and therefore should be persisted together. Repositories encapsulate persistence logic, separating domain behaviour from storage implementation details.

Together, these patterns constrain how and where data can be modified. Persistence operations originating in one subsystem would no longer traverse arbitrary model relationships. Instead, access would be mediated through aggregate boundaries and integration layers, restoring separation between subsystems.

Importantly, this approach would allow the Django ORM to remain in place behind a defined persistence boundary. Existing database structures would not need immediate reorganisation. Over time, direct inter-model dependencies could be refactored behind repositories and replaced with richer domain representations.



Change Isolation with Bounded Contexts





At the HTTP boundary, automatic persistence through DRF's default serialisation layer would be removed. Instead, custom serialisers would construct domain objects without persisting them directly, allowing a service layer to coordinate business logic and delegate persistence to repositories.

To support this architectural shift, targeted design training could be delivered to development teams to reinforce the rationale behind the changes and reduce regression to framework-default patterns. Hiring criteria could be adjusted to prioritise experience in complex domain modelling rather than framework familiarity alone.

Resolving Innovation Constraints

Addressing the Trade-Off in the Training Data Approach

The company's training methodology was a deliberate strategic choice. By enabling models to be trained on a small number of labelled documents, the product reduced the upfront data burden on customers and strengthened its competitive position. However, this approach entailed a structural trade-off: models generalised less well and required customer-specific configuration and skilled labelling to achieve consistent performance.

More general models would have required significantly larger training datasets and greater compute resources, increasing customer effort and weakening the company's market differentiator. For this reason, the training paradigm itself was not considered negotiable.

Given that constraint, the focus needed to be on mitigating its downstream operational effects. Rather than attempting to alter the product's core strategy, intervention could target the surrounding system: simplifying deployment configuration, increasing deployment robustness, improving experimentation ergonomics, reducing friction in moving improvements to production, and streamlining upgrade processes.

These changes would concentrate in the ML pipeline and MLOps control architecture. Customer-specific configuration would remain necessary, but the goal would be to reduce the time, risk, and coordination cost associated with that configuration. The objective was to absorb the trade-off more efficiently within the platform.

Designing an MLOps Control Architecture

The absence of a dedicated MLOps control layer limited visibility, model tracking, and resource control. With orchestration embedded inside the web application, execution logic, configuration, and infrastructure management were tightly coupled.



Introducing a general-purpose containerised workflow engine would segregate orchestration from the application layer. Doing so would result in clearer visibility into training runs, access to logs and metrics during execution, and explicit configuration that could be inspected and adjusted per run. Pipeline components could receive configuration at runtime rather than requiring code changes or application restarts.

Compared to a worker queue such as Celery, a workflow engine would allow pipeline structure and execution order to be defined outside the web application lifecycle. Changing sequences or resource allocation would no longer require modifying backend settings and restarting the application. Running pipelines in containers also enables process isolation and parallel execution. Using an orchestration platform such as Kubernetes could further add automatic scaling and more precise resource control.

Better model management also becomes available under this structure. Instead of storing models implicitly within the application database, versioning and artifact tracking could be explicit. Integrating experiment tracking with the workflow engine allows training runs, parameters, and evaluation metrics to be recorded consistently and surfaced immediately. Saving and serving models in an interoperable runtime preserves flexibility across different machine learning frameworks.

While such a system can operate in different deployment modes, its benefits increase in containerised environments. Supporting multiple deployment types increases operational variance and overhead. Standardising on container-based orchestration would reduce configuration drift and simplify scaling.

Separating orchestration, model management, and infrastructure control from the web application would enable the company to turn experimentation and deployment into infrastructure-managed processes. Doing so would reduce the cost of experimentation, improve observability, and make iteration faster and safer.

Renovating the Legacy ML Pipeline

The company's existing machine learning pipeline had been developed during the startup phase and had not undergone a structured redesign as the company grew. Over time, it accumulated technical debt. It relied on mixed tooling, required careful manual configuration, and was sensitive to changes in its execution environment. Because it was central to the company's product, modifying it carried risk, which reduced willingness to evolve it.

A machine learning pipeline is largely a composition of well-known statistical components implemented across established frameworks. When orchestration, configuration, and environment management are tightly coupled to a single in-house implementation, experimentation becomes con-



strained by the limitations of that implementation. As technical debt increases, each modification requires more effort and carries more uncertainty.

Different ML frameworks offer different trade-offs in usability, performance, and flexibility - and their model artifacts are often incompatible. Committing tightly to a single framework reduces flexibility. Changing frameworks later can be costly and disruptive.

Separating environment control from framework choice reduces this risk. Using a containerised workflow engine allows experiments to run in isolated environments, each configured with the dependencies required by a specific framework. Doing so reduces environment sensitivity and removes the need to standardise on a single package manager. Researchers could select the framework best suited to a given experiment while maintaining consistent execution control - for example, using `fastai` to prototype quickly and iterate on model ideas, and then transitioning to TensorFlow when optimising for large-scale training or production deployment.

Framework incompatibility also affects model artifacts. Different frameworks save models in different formats. Without a common prediction interface, improvements made in one framework cannot be easily reused elsewhere. Introducing an interoperable model format and runtime such as ONNX allows models trained in different frameworks to be deployed through a consistent serving layer. This reduces friction when moving improvements between experiments and production environments.

Separating orchestration from implementation choice, and standardising model interoperability would allow the company to increase the flexibility of their setup while reducing risk. Doing so lowers the cost of experimentation and makes changes to the pipeline safer to introduce over time.

Resolving Difficult Installations and Upgrades

Transitioning all existing customers away from bare-metal deployments was not feasible. In several cases, client compliance policies mandated bare-metal on-premises installations, limiting the company's ability to standardise deployment models.

Under these conditions, improving the bare-metal experience offered limited return on investment. The requirement for environment-specific configuration and manual coordination meant that operational costs remained structurally high. Maintaining bare metal as an equal deployment path ensured ongoing high cost and complexity for both the company and the customers who required it.

A shift in development focus toward containerised, cloud-based deployment would change the economic balance. Concentrating primary development effort on standardised, container-based environments would reduce operational variance and improve scalability. Bare-metal deployments



African Pelagic

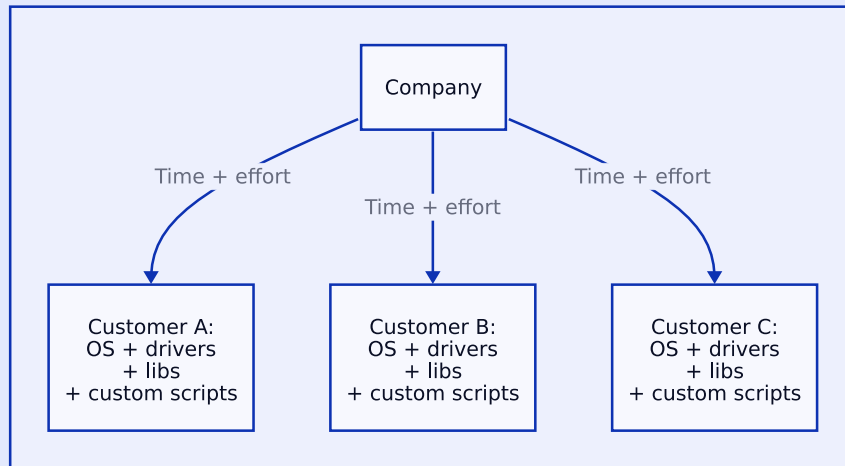
could remain supported, but treated as exceptions rather than the default path, with pricing structures that reflected the additional operational burden of bespoke deployments to align cost with complexity.

Over time, as containerisation and cloud infrastructure become industry norms, standardised deployment models will tend to offer increasing efficiency advantages. As market expectations shift, organisations with strict deployment constraints will face growing pressure to align with prevailing operational standards. In such an environment, the proportion of customers requiring bespoke bare-metal installations is likely to decline relative to the broader market.

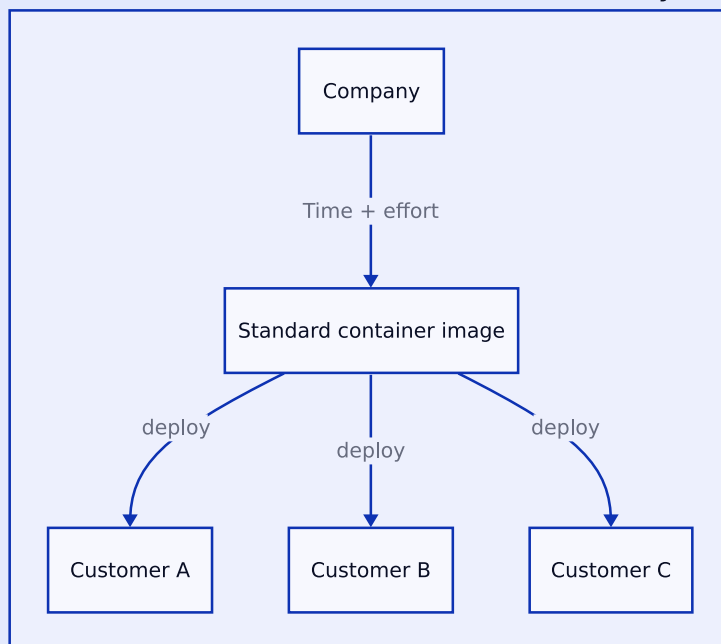


Deployment Models

Bare metal: Effort scales linearly



Containerised: Effort scales horizontally





Outlook

Trajectory Without Change

If growth continues under the current structure, the existing constraints are likely to intensify rather than stabilise.

With domain models remaining tightly coupled, each new feature will require coordinated changes across shared persistence structures. Bottlenecks around core models will become more frequent, and more time will be spent resolving conflicts and retesting shared components. Adding engineers will increase coordination overhead without proportionally increasing delivery.

With orchestration embedded inside the web application, experimentation will continue to depend on code changes and restart cycles. As complexity grows, iteration will slow and fragility will increase. A larger share of research time will shift toward stabilising deployments rather than improving models.

If bare-metal deployment remains a deployment primary model, operational variance will expand with each new customer environment. Time spent on installation and upgrade cycles will increase, support load will rise, and consulting effort per customer will grow without concomitant return.

Over time, bottlenecks multiply, fragility increases, and more effort is spent maintaining stability than delivering improvement. Under these conditions, company growth becomes increasingly costly.

Trajectory With Change

Changing the system's trajectory would require redefining boundaries rather than refining existing processes. Improvement would come from altering how responsibilities and dependencies are organised.

Introducing clear domain boundaries and controlling persistence coupling would reduce the number of subsystems affected by a single modification, lowering coordination overhead and improving predictability.

Separating pipeline orchestration and model management from the web application would increase capacity for innovation. A distinct control layer would shorten restart cycles, improve observability, and allow experimentation without destabilising deployments. Research effort could shift toward advancing model capability.

Operational efficiency would improve if deployment environments converged on a containerised ap-



African Pelagic

proach, by reducing configuration drift and making installation and upgrade behaviour more consistent. Over time, efficiency gains would accumulate.

These changes would contain project complexity. Growth would be manageable as effort would increasingly reduce friction and fragility.



Conclusion

This case reflects a common dynamic in scaling technical organisations.

Early-stage market demands favour speed of delivery over separation of concerns. Default configurations and implicit coupling reduce friction in the short term. As the system grows, those same choices increase coordination cost.

Rising coordination cost influences hiring, process, and deployment decisions. Hiring prioritises framework familiarity; process converges on convention; deployment models avoid structural change. These adaptations stabilise short-term delivery while reinforcing the original architectural coupling.

Over time, the organisation becomes structurally constrained by path dependence. The mechanisms that enabled rapid early growth begin to limit further scale.

The pattern is not unique to a specific framework, technology, or industry. It arises wherever early optimisation for speed precedes explicit boundary design.

Recognising this feedback loop early enables adjustment before coordination costs compound. Failing to recognise it ensures reduced agility, innovation, growth, and return.



African Pelagic

contact@africanpelagic.com
<https://africanpelagic.com>