WHITE PAPER

Customise ERPNext Safely: Custom Fields, Custom Apps & Upgrade-Proof Change

How to bend ERPNext to your business without painting yourself into a corner — the Frappe customisation ladder, from no-code fields to a proper custom app, and why forking core is the one road to avoid.

For CTOs, IT leaders & ERPNext admins · **9 min read**

Acube Innovations · Official ERPNext Partner · acubeinnovations.com · +91 62358 66111

EXECUTIVE SUMMARY

The best thing about ERPNext is that you can change almost anything. The most expensive mistake teams make is changing it the wrong way. ERPNext runs on the Frappe framework, which offers a deliberate ladder of customisation mechanisms — from no-code custom fields and property setters, through light client and server scripts, up to a proper version-controlled custom app that hooks into the system without touching it. Used correctly, every one of these survives an upgrade untouched. The trap is the shortcut: editing ERPNext's core files directly, or forking the codebase. It works on day one and turns every future upgrade into a merge-conflict nightmare — the single most common reason a business gets stranded on an old, unsupported version. This paper maps the safe ladder, explains what belongs on each rung, and lays out a governance policy so your customisations stay an asset instead of a liability.

Why bad customisation breaks upgrades — and good customisation doesn't

ERPNext is open source, so nothing stops a developer from opening a core Python or JavaScript file and editing it in place to get the behaviour a business wants. It works immediately, which is exactly why it's so tempting — and exactly why it's a trap. The moment you modify a file that the framework itself ships and maintains, you've forked the truth. When the next ERPNext version changes that same file, your edit and the upstream change collide, and someone has to hand-merge them. Multiply that across dozens of tweaks accumulated over a couple of years and the upgrade stops being a routine command and becomes a project nobody wants to fund. This is the number-one reason businesses get stranded on an old, unsupported ERPNext version — not because upgrading is hard, but because their customisations were done in a way that fights the upgrade.

The good news is that Frappe — the framework ERPNext is built on — was designed to make this avoidable. It provides a spectrum of customisation mechanisms that live outside the core code: some stored as records in your own database, others packaged in a separate app you control. Because none of them touch the files ERPNext ships, an upgrade flows straight past them. The skill isn't avoiding customisation — ERPNext is meant to be customised. The skill is choosing the mechanism that gets the change you need while keeping it on the upgrade-safe side of the line.

- The danger isn't customising — it's customising by editing core files ERPNext ships and maintains.
- Every core edit is a future merge conflict; enough of them and upgrades quietly become unaffordable.
- Frappe provides upgrade-safe mechanisms by design — stored as your own records or in your own app.
- The whole discipline is picking the right rung on the ladder for each change.

The ERPNext customisation ladder — safest to riskiest

1. **Customize Form / Custom Field / Property Setter**
   no code; stored as your own records; the right answer for most changes.

2. **Client Script**
   light browser-side logic (validations, dynamic fields, live calculations); stored in the database.

3. **Server Script**
   authoritative Python on events, APIs or schedulers; sandboxed; enforced no matter how data arrives.

4. **Custom App with hooks & overrides**
   your own DocTypes, logic and integrations, version-controlled; extends core without touching it.

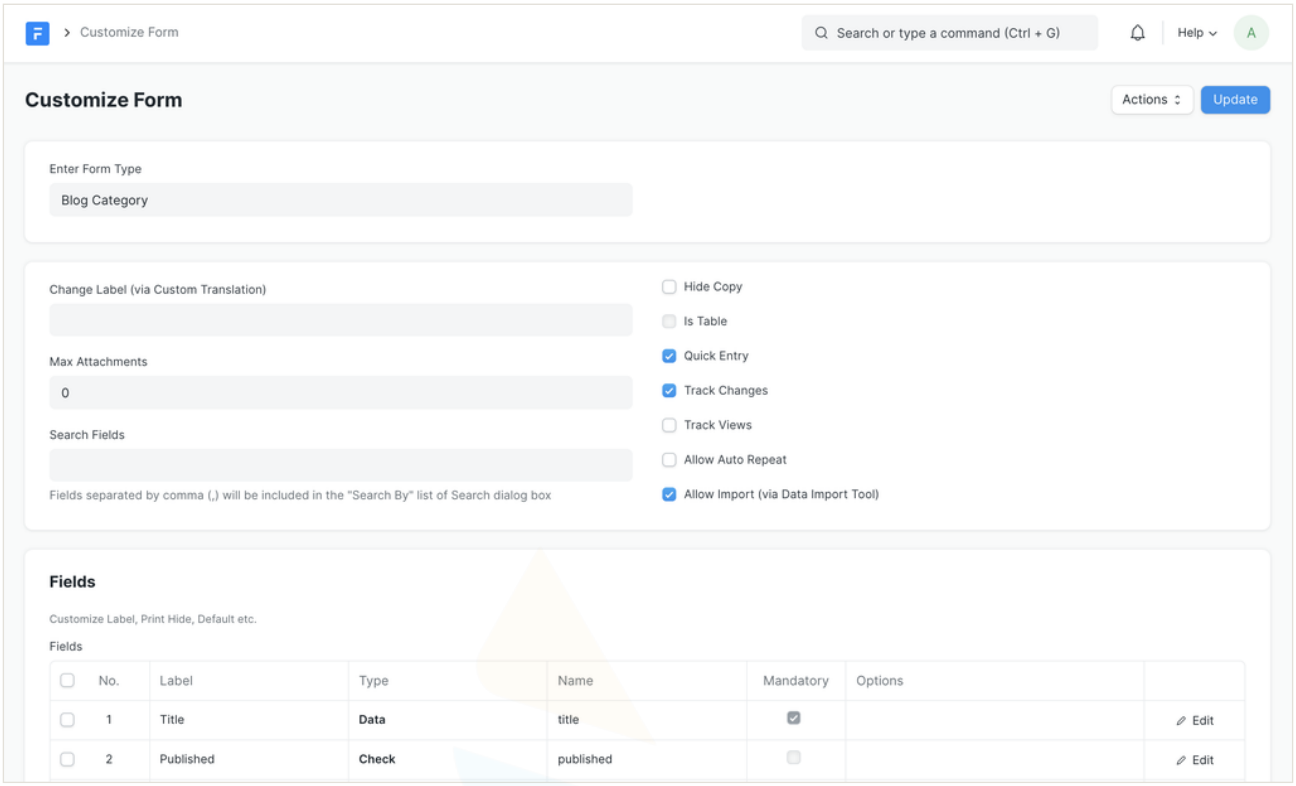
5. **Editing / forking core ERPNext**
   AVOID. Every upgrade becomes a merge conflict; unsupported; the road to being stranded.

The safe foundation: Customize Form, Custom Fields and Property Setters (no code)

The lowest, safest rung needs no code at all. Through the Customize Form view, a System Manager can reshape almost any standard DocType — hiding fields, making them mandatory or read-only, renaming labels, changing options, reordering the layout — and add entirely new fields to a form. Critically, none of this rewrites the DocType that ERPNext ships. Frappe stores your changes as separate records: a Property Setter for each property you override (a hidden flag, a mandatory flag, a new label), and a Custom Field record for each field you add. At runtime the framework layers these over the standard definition when it builds the form, so you see your customised version while the original ships untouched underneath.

Because the customisations are their own records rather than edits to core, an upgrade simply re-applies them over the new version — nothing to merge, nothing to break. And because Property Setters and Custom Fields are ordinary DocTypes, they can be exported as fixtures inside a custom app and version-controlled, so the exact same set of fields and tweaks deploys identically to your test and production sites. For the large majority of what businesses actually want — an extra field on the Sales Order, a mandatory cost centre, a hidden section, a friendlier label — this no-code layer is the complete and correct answer. Reach for code only when configuration genuinely can't express the requirement.

- Customize Form reshapes standard DocTypes — hide, reorder, relabel, mandatory, read-only, options.
- Custom Field adds new fields to a standard form without editing the shipped DocType.
- Changes are stored as Property Setter and Custom Field records, layered on at runtime via the metadata.
- Upgrade-safe by construction — and exportable as fixtures for version-controlled, repeatable deployment.



Customize Form in ERPNext — reshape a standard DocType's fields and properties without editing any core file; changes are saved as Property Setters and Custom Fields.

The next rung: Client Scripts and Server Scripts (light code, still upgrade-safe)

When configuration alone can't do it — you need logic, a calculation, a conditional validation, an auto-populated value — Frappe's next rung is scripting, and it stays firmly on the safe side of the line because scripts are stored as records in your database, not as edits to core files. A Client Script runs in the browser: it hooks into form events to validate input, set or clear field values, show or hide fields dynamically, fetch data from linked records, or compute a figure as the user types. Because it lives in the database, an ERPNext upgrade never overwrites it.

The important nuance is where a rule is enforced. A Client Script only fires in the standard browser form — so a validation written there protects users clicking through the desk, but not data coming in via the API, imports or the system console. When a rule must hold everywhere, you use a Server Script: Python that runs server-side on document events (before or after save, submit, cancel and so on) or as a custom API endpoint, scheduler job or permission query. Server Scripts execute in a restricted, sandboxed Python environment for safety, and on shared or hosted setups they may be disabled by default and need enabling. As a rule of thumb: use a Client Script for interface behaviour and instant feedback, and a Server Script when the logic must be authoritative and enforced no matter how the data arrives.

- Client Script — browser-side form logic: validations, set/hide fields, fetch from links, live calculations.
- Server Script — Python on document events, APIs, schedulers or permission queries; runs in a restricted sandbox.

- Client-side rules only apply in the desk form; enforce anything that must hold via a Server Script.
- Both are stored as database records, so upgrades leave them completely intact.

The proper way to build: a custom app with hooks and overrides

Configuration and scripts cover an enormous amount, but serious, structured extension — new DocTypes of your own, complex business logic, integrations, reusable modules, reports and dashboards you want versioned and code-reviewed — belongs in a custom app. This is the professional's default and the mechanism Frappe most wants you to use for anything non-trivial. You create it with a single bench command (bench new-app), and it lives as its own directory in the bench, separate from ERPNext and Frappe, with its own Git history. You install it onto a site alongside ERPNext; you never modify ERPNext to accommodate it.

The key to a custom app is hooks.py — the file where you 'hook into' framework events and extend or override standard behaviour without touching a single core file. Through it you can respond to document lifecycle events across any DocType (doc_events), override the class that powers a standard DocType (override_doctype_class), replace a standard API method with your own (override_whitelisted_methods), add scheduled jobs, inject client scripts, and more. You can even export your Custom Fields and Property Setters as fixtures within the app, so your entire customisation — code, config and all — deploys as one version-controlled, testable unit. Because everything ERPNext ships stays pristine, upgrading ERPNext and upgrading your app are two independent, clean operations. This is what 'upgrade-proof customisation' actually looks like in practice.

- A custom app is a separate, Git-versioned package installed alongside ERPNext — core is never edited.
- hooks.py extends and overrides standard behaviour: doc_events, override_doctype_class, override_whitelisted_methods, schedulers.
- House your own DocTypes, business logic, integrations, reports and dashboards there — code-reviewed and testable.
- Bundle Custom Fields and Property Setters as fixtures so the whole customisation deploys as one unit.

Why forking core ERPNext is a trap

There is one road that looks like a shortcut and turns into a dead end: forking ERPNext or Frappe and editing the core directly. It's seductive because the change is right there — open the file, change the line, done. But you've now taken ownership of code the framework's authors also own and keep changing. Every future ERPNext release that touches your edited files arrives as a merge conflict you must resolve by hand, testing carefully each time to be sure you haven't reintroduced a bug or lost your change. The maintenance burden compounds release over release until upgrading is so painful that it simply stops happening — and a business running years-old, unsupported ERPNext loses security fixes, new features and, eventually, the ability to get help.

There's a support dimension too: modified or forked versions of the framework are not supported, so you also forfeit the safety net. The honest rule is that almost anything a fork could achieve can be achieved instead through a Custom Field, a script, or a hook or override in a custom app — the

upgrade-safe mechanisms exist precisely so you never need to fork. In the rare genuine edge case where the framework offers no extension point, the right move is to contribute the hook upstream or engineer around it in your app, not to cut the branch you're sitting on. Treat 'edit core' and 'fork' as off-limits, and keep every customisation on the safe side of the line.

- Editing or forking core makes you the owner of code upstream keeps changing — every release becomes a merge conflict.
- The maintenance cost compounds until upgrades stop, stranding you on an old, unsupported version.
- Forked/modified framework versions aren't supported — you lose the safety net as well as the upgrades.
- Nearly anything a fork could do is achievable via a field, script, hook or override — so forking is never the right first answer.

A governance policy for changes

Staying upgrade-safe is less about heroics and more about a simple, enforced policy — because customisation debt accumulates one 'quick fix' at a time, usually made directly in production under deadline pressure. A little governance keeps the whole estate clean.

Adopt a default order of preference: reach for the lowest, simplest rung that solves the problem, and only climb when it genuinely can't. Configure through Customize Form before you script; script before you build; build a custom app before you ever consider anything closer to the core; and treat editing or forking core as forbidden. Make one rule non-negotiable — no customisation, of any kind, is edited directly in production. Build and test it on a separate staging bench first, capture it as code or fixtures in your custom app, and deploy it the same repeatable way every time. Keep a living register of every customisation so nothing is a mystery at upgrade time, and rehearse upgrades on staging before touching production. None of this is heavy; it's the difference between an ERP that grows with you and one that quietly ossifies.

- Default to the simplest rung: configure before you script, script before you build an app, and never edit core.
- No customisation is ever made directly in production — build and test on a staging bench first.
- Capture everything as code or fixtures in your custom app so deployments are repeatable and reviewable.
- Maintain a register of all customisations and rehearse every upgrade on staging before go-live.

Getting help

Most teams can handle the everyday rungs of this ladder in-house — a well-trained admin can add custom fields, adjust properties and write a straightforward client script confidently. Where an experienced partner earns their fee is higher up and at the seams: designing a custom app the right way from the start, deciding when a hook or an override is the correct tool, keeping business logic testable, and setting up the staging-to-production workflow and fixtures so your customisations are a version-controlled asset rather than tribal knowledge. Getting that architecture right once, early, is far cheaper than untangling a pile of production hot-fixes later — or worse, discovering at upgrade time that someone quietly edited core.

As an official ERPNext partner working with Indian businesses, we build customisations the upgrade-safe way by default: no-code where it suffices, scripts where they fit, and a clean custom app with proper hooks for everything structured — all deployed through a disciplined staging workflow. The result is an ERPNext that molds to how you actually work and still upgrades in an afternoon, not a quarter. If your system has drifted toward core edits or a fork, we can also help you refactor those changes back onto the safe ladder before they cost you a version.

KEY TAKEAWAYS

1. ERPNext is built to be customised — the risk is never customising, it's editing or forking core files, which turns every future upgrade into a merge conflict.
2. Start at the no-code rung: Customize Form, Custom Fields and Property Setters cover most needs and are stored as your own records, so upgrades leave them intact.
3. For logic, use Client Scripts for browser-side behaviour and Server Scripts when a rule must be enforced everywhere; both live in the database, not in core.
4. For anything structured, build a version-controlled custom app that extends ERPNext through hooks.py and overrides — the proper, upgrade-proof way.
5. Enforce a governance policy: simplest rung first, never edit in production, capture everything as code or fixtures, keep a register, and rehearse upgrades on staging.

FAQ

Can I customise ERPNext without breaking future upgrades?
Yes — that's exactly what the Frappe framework is designed for. Custom Fields, Property Setters (via Customize Form), Client Scripts, Server Scripts and a custom app with hooks all live outside the core files ERPNext ships, either as records in your database or in your own version-controlled app. Because none of them edit core, an upgrade flows straight past them. The only thing that breaks upgrades is editing or forking the core code directly.

What's the difference between a Client Script and a Server Script in ERPNext?
A Client Script runs in the browser and handles form-level behaviour — validating input, setting or hiding fields, fetching from linked records, live calculations — but it only fires in the standard desk form. A Server Script runs server-side in a restricted Python sandbox on document events, APIs, schedulers or permission queries, so its logic is enforced no matter how the data arrives (desk, API, import or console). Use a Client Script for interface feedback and a Server Script when a rule must be authoritative.

When should I build a custom app instead of using custom fields and scripts?
Use the no-code and scripting rungs for fields, property tweaks and self-contained logic. Move to a custom app when you need structured, versioned extension: your own DocTypes, complex or reusable business logic, integrations, custom reports and dashboards, or overrides of standard behaviour. A custom app is created with bench new-app, installed alongside ERPNext, and extends the system through hooks.py without touching core — so it's fully upgrade-safe and code-reviewable.

Why shouldn't I just edit ERPNext's core code or fork it?
Because you'd be taking ownership of code the framework authors also maintain and keep changing. Every future release that touches your edited files arrives as a merge conflict you must resolve and re-test by hand, and the cost compounds until upgrading becomes so painful it stops — stranding you on an old, unsupported version without security fixes or new features. Modified or forked framework versions also aren't supported. Almost anything a fork could achieve can instead be done with a custom field, a script, or a hook or override in a custom app.

Talk to a real ERPNext expert.
Call or WhatsApp +91 62358 66111 · info@acube.co · acubeinnovations.com