

How the system is built, and what keeps it correct.

For the engineer or the CTO who has to say yes. What every a-systems build is made of, how its data stays correct, how it is protected, and how a change reaches your live system. Each figure was measured on a running system, and the text says where.

A-SYSTEMS.PRO/DOCS/TECHNICAL-OVERVIEW · EDITION OCTOBER 2026 (V.1) · UPDATED EVERY MONTH

CONTENTS

- 01 In one page
- 02 Principles
- 03 Architecture
- 04 One foundation, one build per client
- 05 The stack, and the way out of each part
- 06 Speed on real volumes
- 07 Data that stays correct
- 08 Security
- 09 AI assistants, through the same door
- 10 Integrations
- 11 How a change reaches production
- 12 Operations and failure

The system on one page.

What it is	One system for a company's sales, purchasing, warehouse, delivery, invoicing, HR, production, service and accounting. It replaces the ERP you already have one business flow at a time, and accounting moves last.
How it is built	Every client gets a build of its own: its own code repository, its own database, and its own server once the system is accepted. Every build stands on one shared foundation, so a fix made in the foundation reaches every build.
Stack	TypeScript end to end. Bun runs it, Hono serves the API, Next.js draws the screens, PostgreSQL holds the data through Prisma, background jobs wait in the same database, and Docker runs it all.
Where it runs	While we build: on our servers in the European Union. After acceptance and full payment: on your own infrastructure, wherever you choose.
Size of the reference build	280+ data entities, 850+ API routes and 210+ screens in 19 modules (counted in the code on 16 October 2026)
Speed	On a database of 1.3 million+ rows shaped like a distributor's live system, the slowest list answers in about 140 ms against a budget of one second (measured 16 October 2026)
Correctness	Money is stored in whole cents. Every posted entry is sealed to the one before it. The database itself refuses to change a posted entry.
Checks	10,000+ automated tests on the reference build and its foundation, run against a real PostgreSQL database; every one must pass before a change can merge. 780+ browser scenarios walk the screens the way a person does.
AI	An AI assistant such as Claude or ChatGPT connects as a named person, with that person's role, through the same API the screens use.

Six rules every part follows.

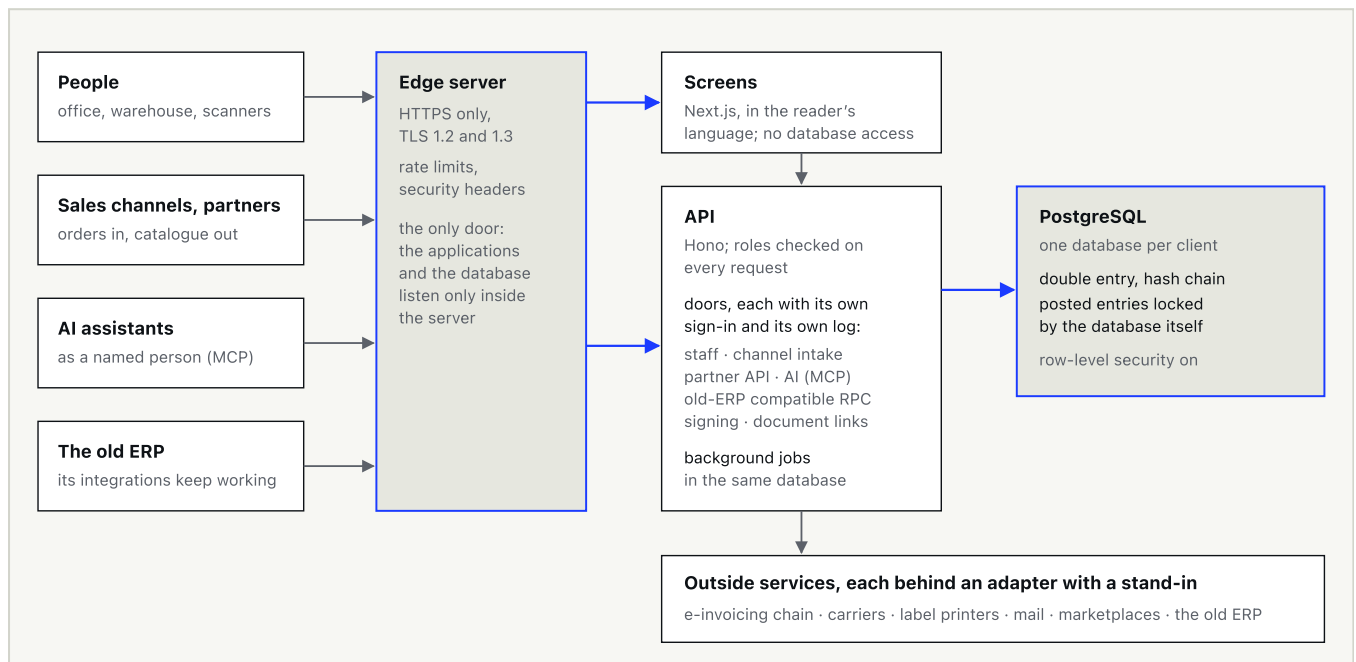
- + **The screens never touch the database.** All business logic lives in the API. So a rule exists in exactly one place, and a script has to obey it just like a person on screen.
- + **One step simpler than the obvious answer.** One PostgreSQL database per client, one API process, background jobs in the same database, Docker Compose instead of Kubernetes. Every extra layer costs something in every month of operation, not once at installation.
- + **Every part of the stack has a boring way out.** Replacing any part costs a configuration change, not a rewrite. Section 05 names each one.
- + **Rules that must never be broken live in the database.** PostgreSQL itself refuses a change to a posted entry and blocks every way into the data except through the application. A script, a console session or a future developer cannot get round them by not knowing about them.
- + **Anything that runs by itself must say when it stops.** Monitoring asks "is it doing its job", not "is the process alive".

- + A written specification before the code, and a failing test before the code that moves money, tax or stock. Section 11 shows how the release process enforces both.

03 · ARCHITECTURE

One way in, and the database at the end of it.

Every request takes the same road: through the edge server, into the API, and only then to the data. Neither the screens nor any outside system reaches the database directly.



The screens ask the API, the API asks the database. Outside services are reached from the API through adapters, never the other way round.

Doors into the API

The API has one door per kind of visitor, and each door has its own sign-in and its own log.

- + **Staff:** the screens, the warehouse scanners and phones, with the person's session and roles.
- + **Channel intake:** orders from sales channels and marketplaces. It is idempotent (the same order sent twice is one order), and it keeps a log of everything received, so any event can be replayed.
- + **Partner API:** catalogue and offers out to a marketplace or a partner, one token per partner.
- + **AI door:** assistants such as Claude or ChatGPT, signed in as a named person (section 09).
- + **Old-ERP compatible RPC:** integrations written for an Odoo-style XML-RPC or JSON-RPC interface keep working during the move, without being rewritten.
- + **Signing and document links:** an outside party signs a document, or opens a shared file, at a link that expires. The link opens that one document and nothing else.

Background work waits in a queue inside the same PostgreSQL database (pg-boss). Writing a document and queuing its follow-up job happen in one transaction, so there is never a job without its document or a document without its job. A request that would take longer than two seconds becomes a background job, never a slow screen.

One foundation, and a build of its own for each client.

We sell one product, built more than once. What every build shares lives in one place, the foundation. What is specific to one client lives in that client’s build and nowhere else.

The foundation

The foundation’s core is a versioned package of pure business logic: no database client, no web framework, no runtime of its own, so every build can use it whatever else it runs. It holds the rules that must behave the same everywhere: stock valuation, putaway and removal, landed cost, barcode scanning, the ledger, budgets, approvals, CRM, HR and payroll, electronic signature, currency conversion, carriers, order intake, document versions, translations and the AI door.

- + **One rule in one place.** A fix to a rule in the foundation reaches every build with its next version. A build may never import another build’s code; a check at the start of every deploy refuses one that does.
- + **Each build moves on purpose.** A build takes a new foundation version as a pinned package, and its whole test suite runs against it before it ships. Nothing changes under a build that did not ask for it.
- + **No registry, no outside credential.** The package travels inside the build’s own repository. No server that runs a client system needs access to our code hosting.
- + **What you receive.** The code written for your business is yours. The foundation’s core comes as a compiled package under a licence with no expiry and no user count, and its source code goes into escrow after delivery.

The build

A build is described by a profile: which modules it carries, which add-on packs (label printing, a carrier, cameras at the packing desk), which connections to outside systems, and which book it keeps (currency, country, time zone). Every file in the code base is marked as foundation, shared template or specific to one client, and a check refuses a file with no mark.

A client’s own server is deployed from a repository generated for that client: the parts of the build that client uses, and nothing that belongs to anyone else. The deploy runs from a tagged version, dumps the database before it migrates, and keeps the previous images to roll back to.

The stack, and the way out of each part.

The stack is fixed in the project’s written rules and changes only by an explicit decision. What matters more than each choice is that each one has an exit written down in advance.

LAYER	CHOICE	WHY	THE WAY OUT
Runtime	Bun 1.4, TypeScript in strict mode	Fast, and one tool to run, test and install	Node LTS: Bun-only APIs are banned in the server and business code, and a test checks it on every change
API	Hono 4	Small, independent of the runtime, typed end to end	Runs on Node as it is

Screens	Next.js 16, Tailwind, TanStack	The industry standard; server rendering and fast lists on large volumes	Plain React: the screens hold no business logic to move
Database	PostgreSQL 16, Prisma 7	One database for all data, a typed schema, declared indexes, migrations reviewed like code	Any PostgreSQL host: moving it is one connection string
Queue	pg-boss, in the same database	One service fewer; the job and its data survive a restart together	Redis and BullMQ, a configuration change, if a measurement ever shows the need
Deployment	Docker Compose and one script from the repository	Nothing on a server is installed by hand; a new server is Docker plus the deploy	Any host that runs Docker

The system is not tied to Bun, to a hosting provider, or to a managed database. It runs anywhere Docker runs.

06 · SPEED

Speed measured on real volumes, not on demo data.

Before a build reaches production, we read the volumes out of the client's live system and generate a database of the same size and shape. Then we measure every list over HTTP: the time until the answer reaches the screen, with its filters, counters and totals, not a bare database query.

On the reference build that database held **1.3 million+ rows**: products, customers, orders and their lines, purchase orders, transfers, stock and the full movement ledger. The budget for any list is one second, written into the project's rules as a property of the product.

SURFACE, PRODUCTION VOLUME	ANSWER
Product list, one page of 92,000+	20 ms
Product search by name	42 ms
Customer list with counters	17 ms
Order list with its total	13 ms
Order list, page 10 (deep paging)	12 ms
Orders grouped by customer	42 ms
Purchase order list with its total	14 ms
Movement ledger, one page of 867,000+	36 ms
Vendor prices, one page	6 ms
Stock on hand, one page	142 ms
CSV export of all 54,000+ transfers	240 ms

Measured on 16 October 2026 against the current code, over HTTP: the median of three runs, each the best of five after a warm-up. The slowest list, stock on hand, takes about a seventh of the budget; the heaviest operation, exporting every transfer to CSV, a quarter.

What the rehearsal is for

The only surface that ever broke the budget was not one a person sees: an internal stock check compared every balance in memory and took almost twelve seconds. It moved to a nightly job and its answer became three milliseconds. That is the point of measuring before production: the limits are found in rehearsal, not by users.

Headroom

The reference build's working data fits in the memory of a modest server with an order of magnitude to spare, so growth does not mean buying hardware in advance. Each step up is tied to a signal that can be measured: a bigger server when the processor stays above 60%, the queue moved out of the database if jobs start to compete with people, a read replica if heavy reports slow the warehouse, a second application server when one is not enough. The API keeps no state of its own, so scaling out means adding machines, not redesigning.

07 · DATA INTEGRITY

Data that stays correct, even when someone makes a mistake.

A bigger server fixes slowness. It does not fix a wrong number in the ledger. These are the mechanisms that keep numbers right.

- + **Money is a whole number.** The ledger stores amounts as integer cents. Debits equal credits exactly, not "within a tolerance", and no amount ever passes through floating point. The classic rounding bugs cannot happen.
- + **Double entry with a hash chain.** Every posted entry stores a cryptographic fingerprint of itself and of the entry before it, per journal. Change a figure after the fact and the chain breaks, and the nightly check finds it.
- + **The database refuses changes to what is posted.** PostgreSQL triggers, not only application code, forbid editing a posted entry. A rule inside a service protects only the paths that go through that service. A rule in the database protects every path. A correction in accounting is always a reversal, never a delete, and lock dates stop anyone rewriting a closed period.
- + **Two people, one document.** Sales orders and invoices carry a version that the database moves on every write. When two people edit the same document, the second save is refused with the name of the person who changed it and when, and what was typed stays on screen to apply again. Nothing is silently overwritten.
- + **The business date is separate from the typing date.** Every document that moves stock or revenue carries the date it takes effect, apart from the moment it was entered. Reports use the business date by default.
- + **Nightly checks of the books against the documents.** Every night the ledger balance, the stock valuation and the control accounts are reconciled against the documents behind them, and any difference becomes a task with a name on it. Stock on hand must agree with the movement journal, and this is checked, not assumed.

- + **Backups are read back and leave the machine.** Each daily backup is read back with the restore tool right after it is written, so a broken file is found the same night. Copies stay on the server for seven days and on a second machine, in a different failure domain, for thirty. A backup that did not run or did not arrive raises an alert.
- + **Restore is tested by doing it.** On 8 October 2026 nine databases were restored in full into empty databases with no network: eight from the offsite copy of our stand and one from a client's own server. There were no errors, and every table matched the live database as it stood at the moment of the backup. The drill also found that the backups lacked the database roles a restore needs; they carry them since.

08 · SECURITY

Security that is built in, not added on.

These are properties of the running system. What we do not have yet (an external penetration test, ISO 27001, disk encryption on our own servers) is listed with its status on the [Trust page](#), and this document does not repeat it.

Signing in

- + Passwords are stored only as salted script hashes. A password exists in clear text nowhere and is never sent by email.
- + Every failed sign-in (unknown address, disabled account, wrong password) gets the same answer in the same time, so nobody outside can find out which accounts exist.
- + Five failures in a row lock the account for fifteen minutes, and the edge server limits attempts before they reach the application.
- + A second factor is available to every person: an authenticator app or a six-digit code by email. A used code never works twice, and an administrator sees who has it on and can reset it. It is required for AI access.
- + The session lives in a cookie that page scripts cannot read (HttpOnly, SameSite) and ends after twelve hours. A person switched off loses access at their next request, because roles are checked on every request.

Who may see and do what

- + Roles (sales, purchasing, logistics, finance, quality, administration, read-only) are checked by the API on every route, not by hiding buttons. A person can hold more than one.
- + Rights decide what is sent, not only what is shown: cost and margin never leave the server for a person not entitled to them.
- + Every change to a setting is written to an audit log with who and when. This happens in the platform, so a new settings screen is audited without anyone remembering to add it.

The perimeter

- + The edge server is the only door: TLS 1.2 and 1.3 only, HSTS, a Content-Security-Policy that is enforced, a referrer policy, rate limits, and no software versions announced.
- + The applications and the database listen only inside the server. The database has no port on the internet, so it cannot be scanned and no password can be guessed against it. Row-level security is on for every table, and a test fails the build if a table is ever left out.

- + Server access is by key only, with password login disabled. Repeated failed attempts are banned automatically, the firewall refuses everything that is not the web or the key-only door, and security updates install themselves.

The code

- + Every database statement goes through the ORM with its values bound. Raw SQL is allowed only for what the ORM cannot say (a row lock, an aggregate), and each file that holds it is recorded with its reason: the merge gate and the deploy refuse one that is not.
- + Uploads are limited by size and type, and a file's identifier is never enough to open it. No screen turns off the framework's HTML escaping.
- + Library security advisories arrive automatically, and every change gets an automated code review before it merges.

Test systems cannot do harm

- + A system sends nothing to a client's customers until the client switches that on. Until then every letter is redirected to a test address and marked as such.
- + The public demos run on invented data and rebuild themselves. A check on every deploy confirms that neither their data nor their pages contain anything that belongs to a real person or company.

09 · AI ACCESS

AI assistants, through the same door as people.

An assistant such as Claude or ChatGPT can work in the system for a person: find an order, read the stock, prepare a document. It does so as that person, never as a robot with a master key.

- + **It signs in as the person.** The AI door is an MCP endpoint (MCP is the open protocol assistants use to call tools). The assistant connects through OAuth 2.1 with PKCE, the person approves it with their password and second factor, and from then on it acts with that person's roles and nothing more.
- + **Its tools are the screens' own routes.** Every tool calls the same API route a screen calls, so every rule, role check and audit entry applies to the assistant exactly as to a person.
- + **What cannot be undone waits for a person.** An act that cannot be reversed, such as posting or sending, is prepared by the assistant and waits for the person to confirm it in the system.
- + **It leaves a trail.** Every change it makes is recorded under the person's name with the assistant it came through, so a history shows who did what, and how.
- + **It is off until you turn it on.** AI access is enabled per company by an administrator, and each person can disconnect an assistant at any moment.

10 · INTEGRATIONS

Every outside system behind an adapter.

Each connection to the outside world is an adapter with at least two forms: the real one and a stand-in. The whole system runs end to end with every adapter in stand-in mode, which is how it is tested and how a new stand starts. Credentials live only in the server's environment, never in the code.

- + **The national e-invoicing chain.** Every outgoing e-invoice file is built by the system and checked against the tax authority's official schema, and the merge gate runs that check on every change.
 - + **Carriers.** Booking a shipment with the client's own carrier account, customs data, cancellation, and the label printed at the packing desk.
 - + **Label printers.** Through a cloud print service to the printers at each station, with templates in the printers' own language, and a sweep that reads back what each printer actually did.
 - + **Sales channels and marketplaces.** Orders come in through the channel intake door; catalogue and offers go out through the partner API.
 - + **The ERP you have today.** While a flow has not moved yet, the system follows the old ERP read only: catalogue, stock and orders arrive every few minutes, and nothing is written back. Each connection reports when it last succeeded, and an alert fires when that is too long ago.
-

11 · RELEASE PROCESS

How a change reaches production.

Discipline that lives in a document is forgotten under deadline pressure. So each of these is a check that stops the release, not a habit.

- + **A specification first.** Every change starts as a written specification with acceptance criteria. Each one lives in the repository beside the code it describes, and the code follows it, not the other way round.
 - + **A failing test first, for money, tax and stock.** Margins, VAT, invoice files and reservations are written "red before green": the failing test comes before the code. Every new check is first broken on purpose, because a test that cannot fail is worse than no test.
 - + **Contracts over every screen.** A test reads the source of every screen and fails if a new document arrives without its change history, without its links to related records, or outside the shared page layout. The system looks and behaves the same everywhere because a machine checks it.
 - + **The merge gate.** Nobody can change the main code directly, including us. A change goes in only when every check passes: code style, types, the database schema and its migration history, the full test suite against a real PostgreSQL database (not a mock), the build, and validation of outgoing e-invoice files against the tax authority's official schema.
 - + **A guarded deploy.** The one deploy script refuses to run with uncommitted changes, checks that the target server is the one it means (a marker on the server, not a hostname typed by hand), and refuses a release the project's roadmap does not mark as finished. It records the exact version it published, and checks the running system afterwards on real data: does stock reconcile, does every list add up.
 - + **A failed release rolls itself back.** If the new version does not pass its health check, the deploy puts back the version that worked a minute earlier and stops with an error. On a client's own server the database is also dumped before every release.
 - + **Your repository holds only your system.** A client's server is deployed from a repository made for that client: the parts of the build that client uses, and nothing that belongs to anyone else. A check refuses the delivery if it finds another client's name or data.
 - + **The guides come from the tests.** The same browser scenarios that test the system write the step-by-step user guides, with fresh screenshots. An outdated guide is a failing test, so it is visible.
-

Operations, and what happens when something breaks.

Every five minutes a health check asks each part whether it is doing its job: are the background jobs running, did each connection to an outside system succeed recently enough, does stock still agree with its journal, did last night's backup run and arrive offsite, is a certificate about to expire. An alert goes out the moment a check fails, and another when it recovers. Every service runs with a memory limit, restarts by itself, and rotates its logs.

WHAT BROKE	WHAT PEOPLE SEE	WHAT HAPPENS	TIME
One application container	Nothing, or a few seconds of errors	It restarts by itself	seconds
The whole server	The system is down	Restore from a disk image, or rebuild: nothing on a server is installed by hand, so a new one is Docker plus the deploy script	15–40 min
Damaged or deleted data	Records are missing or wrong	Restore into a separate database at the right moment and carry the affected records over. The procedure has been run, not only written	1–3 hours
An outside service (mail, a carrier, the e-invoicing chain)	Some functions take their fallback path	Every outside service has a stub and an alert of its own	–

After handover the alerts go to your people, and the operating manual we hand over (the runbook) is written for someone who has never met us. Who does what before and after handover is on the [Trust page](#).