Athena Accounting

Accounting Rules

Version 1.0.0-rc.1 · AthenaLab

Document date: 2026-08-07 · v1.0.0 (General Availability)

AI-first Malaysian double-entry accounting & e-Invoice

Athena Accounting — Accounting Rules & Posting Model

These rules are the contract the engine must satisfy. They are **non-negotiable** and every phase's tests assert them. §1–§8 are the posting model (implemented in Milestones 1C/1D). **§9 (Chart of accounts & periods) is implemented in Milestone 1B.**

1. Double-entry model

- Every financial event becomes a **journal entry** with **two or more journal

lines**.

- Each line is either a **debit** or a **credit** on exactly one account, for a

non-negative amount.

- **Rule 1 — Balance:** for every entry, `SUM(debits) == SUM(credits)`, exactly,

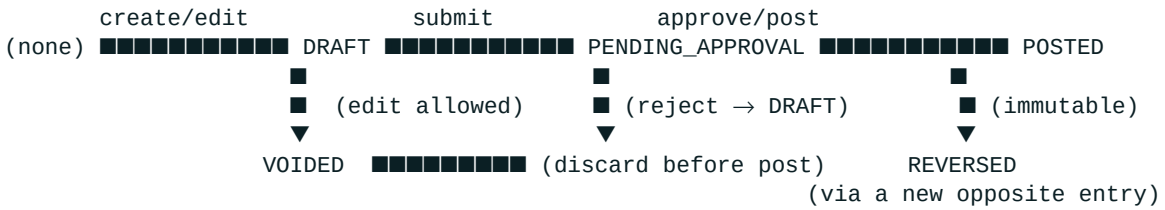
in the entry's currency, to the defined precision. An unbalanced entry can be saved as a **draft** but can **never** reach `posted`.

Account types and normal balances

Type	Normal balance	Debit effect	Credit effect
Asset	Debit	↑	↓
Liability	Credit	↓	↑
Equity	Credit	↓	↑
Revenue	Credit	↓	↑
Expense	Debit	↑	↓

The accounting equation **Assets = Liabilities + Equity** (extended with Revenue – Expense) holds at all times over posted entries; the **balance sheet** and **trial balance** reports assert it.

2. Journal entry lifecycle (status machine)



- **Amounts are always non-negative.** Direction is expressed **only** by the debit

vs credit column — never by a negative number. A negative debit/credit is rejected; a "negative effect" uses the opposite column. Every line has exactly one of debit/credit strictly positive (zero-value lines are rejected), and an entry must have ≥ **2 lines**.

- **No entry is ever hard-deleted.** An abandoned draft is **VOIDED** (retained for

audit); a posted entry is corrected by **REVERSAL**. There is no destructive delete path for ledger data (Readiness Review C3).

- **DRAFT** — editable. May be unbalanced *while being worked on*; cannot post.
- **PENDING_APPROVAL** — awaiting a user with posting authority. Optional per

configuration; some sources (e.g. a manual adjustment by finance) may post directly if the actor has the permission.

- **POSTED** — **Rule 2 — Immutable.** A posted entry's lines, amounts, accounts

and date can never be edited or deleted. Full stop.

- **REVERSED** — a posted entry that has had a system-generated **reversing

entry** created against it. The original stays; the reversal is a new posted entry with debits/credits swapped and a link back to the original.

- **VOIDED** — a draft/pending entry discarded before it ever posted (no ledger

impact; retained for audit).

Corrections

- **Rule 3 — Corrections via reversal/adjustment only.** You never "fix" a posted

entry. You either **reverse** it (then post a corrected new entry) or post an **adjustment** entry. Both are new posted entries with their own audit trail.

---

3. Posting engine invariants

The posting engine (`app/services/ledger`) is the **only** writer of `journal_entries` and `journal_lines`. Every posting call must satisfy:

1. **Rule 7 — Transactional.** A post is one DB transaction. Either the entry,

all its lines, the source-document link, and the audit row all commit, or nothing does. No partial posts.

1. **Rule 1 — Balanced** (see above). Enforced in code *and* by a DB check where

practical; rejected before any write if unbalanced.

1. **Rule 12 — Open period.** The entry's posting date must fall in an **open**

accounting period. Posting into a **locked** period is rejected. (Period lock is how month/year-end close is protected.)

1. **Account posting-eligibility.** Lines may only hit **active**, **postable**

accounts (parent/header accounts are not postable; inactive accounts are not postable).

1. **Rule 5 — Source identified.** Every entry records its `source_type`

(`manual`, `sales_invoice`, `credit_note`, `supplier_bill`, `debit_note`, `payment`, `bank`, `ai_bookkeeper`, `reversal`, `opening_balance`) and `source_id`. No anonymous ledger entries.

1. **Rule 8 & Rule 15 — Idempotent.** Posting is keyed by an **idempotency key**

(source_type + source_id + operation, or a client-supplied key). Re-issuing the same post — including an **e-Invoice retry** or a double-clicked button — returns the *existing* entry and never creates a duplicate. This

is the single most important defence against duplicate financial records.

1. **Rule 4 — Audited.** Every post writes an append-only audit row (actor,

action, entity, before/after summary, correlation id).

1. **Rule 11 — Controlled numbering.** Human-facing document numbers (invoice,

bill, journal, credit note…) come from a per-organization, per-document-type **sequence** with no gaps introduced by the app and no reuse. Numbering is allocated inside the posting transaction.

---

4. Money & currency

- **Rule 9 — Fixed precision.** All monetary values are `Decimal` in Python and

`NUMERIC(18,4)` in SQL. Floats are banned in the accounting path.

- **Storage precision:** amounts stored to **4 decimal places**; presentation

rounds to the currency's minor unit (MYR → 2 dp).

- **Rule 10 — Explicit rounding.** Rounding uses `ROUND_HALF_UP` at explicitly

defined points (line tax, document total, currency conversion). Rounding is never implicit and never done by the display layer to derive a stored figure.

- **Base currency** is set per organization (AthenaLab: **MYR**). MVP is

effectively single-currency; foreign-currency documents store the original amount, the FX rate, and the base-currency amount, and post in base currency. (A full multi-currency revaluation engine is **out of scope** — see `FUTURE_SCOPE.md`.)

- **Tax rounding** follows Malaysian conventions; the specific rule is confirmed

against LHDN guidance in `COMPLIANCE_CHECKLIST.md` before production.

---

5. Source documents → ledger (posting recipes)

Source documents are **not** the ledger. **Rule 6 — reports read posted ledger data**, never invoice/bill totals alone. Approving a source document *posts* it.

Illustrative recipes (final account codes fixed in Phase 1 CoA; tax handling finalised with the e-Invoice/tax review):

- **Sales invoice (approved):**

`Dr Accounts Receivable` (gross) / `Cr Revenue` (net) / `Cr Output Tax` (tax).

- **Credit note (sales):** the reverse of the portion credited.
- **Customer payment:** `Dr Bank` / `Cr Accounts Receivable`, allocated to

specific invoices.

- **Supplier bill (approved):**

`Dr Expense/Asset` (net) / `Dr Input Tax` (tax) / `Cr Accounts Payable` (gross).

- **Debit note (purchase):** reverse of the portion adjusted.
- **Supplier payment:** `Dr Accounts Payable` / `Cr Bank`.

- **Bank reconciliation match:** links an existing posted payment to a bank line;

an unmatched bank line may generate a new entry only through the same posting engine and approval.

- **Rule 13 — Deleting a source document must not destroy ledger history.**

Source documents are **soft-deleted**; any posted entry they produced remains, with its link intact. If a posted document must be undone, it is **reversed**, not deleted.

---

6. AI and posting

- **Rule 14 — AI cannot bypass approval.** The AI Bookkeeper may *propose* a

fully-formed journal entry (accounts, debits, credits, tax) but it is created as a **DRAFT suggestion** that a human must review and approve. AI never calls the posting engine's commit path directly. Full rules in `AI_GOVERNANCE.md`.

---

7. Period close

- Accounting periods have states: **OPEN** → **CLOSED (soft)** → **LOCKED**.
- Posting is allowed only into OPEN periods.
- Closing a period is a controlled action (permission-gated, audited). Re-opening

a LOCKED period is a high-privilege, audited exception.

---

8. Rule-to-test map (asserted from Phase 1 onward)

#	Rule	Where enforced	Test
1	Debits == credits	posting engine + DB check	balanced/unbalanced journal tests
2	Posted entries immutable	ORM guard + no update endpoint	edit-posted-rejected test
3	Corrections via reversal/adjustment	reversal service	reversal test
4	Every txn audited	`write_audit` in posting txn	audit-row-present test
5	Every entry has a source	required columns	source-required test
6	Reports from posted ledger	reports engine reads ledger only	report-vs-invoice test
7	Posting transactional	single DB transaction	partial-post-rollback test
8	Duplicate posting prevented	idempotency key unique constraint	double-post test
9	Decimal money	`NUMERIC(18,4)` / `Decimal`	precision test
10	Explicit rounding	rounding helpers	rounding test
11	Controlled numbering	sequence allocator	numbering-gap/uniqueness test

#	Rule	Where enforced	Test
12	Period locking	posting engine period check	locked-period-rejected test
13	Delete source ≠ destroy ledger	soft delete + retained entries	soft-delete test
14	AI cannot bypass approval	AI writes drafts only	ai-no-autopost test
15	e-Invoice retries don't duplicate	idempotency key	einvoice-retry test

---

9. Chart of accounts & accounting periods (Milestone 1B — implemented)

9.1 Account types & normal balance

- Five account types: **ASSET, LIABILITY, EQUITY, REVENUE, EXPENSE**.
- **Canonical normal balance:** DEBIT for ASSET/EXPENSE; CREDIT for

LIABILITY/EQUITY/REVENUE. The application **rejects** any type/normal-balance combination that violates this — **except** a contra account.

- **Contra accounts** (`is_contra=true`, e.g. accumulated depreciation) carry the

opposite of their type's canonical balance. This is the only sanctioned way normal_balance may differ from the canonical, and it is validated and tested. No automatic contra *posting* behaviour is implemented in 1B.

- `code`, `account_type` and `normal_balance` are **immutable after creation**

(changing them would invalidate history and later posted entries); create a new account instead.

9.2 Account hierarchy

- A child's `parent_account_id` must belong to the **same organization** and the

same account_type.

- An account **cannot be its own parent**; **cycles are rejected** beyond one

level (the ancestor chain is walked: A→A, A→B→A, A→B→C→A are all rejected).

- A **parent that gains a child** has `allow_manual_posting` set to false

(summary/parent accounts are not manual-posting leaves). You cannot set `allow_manual_posting=true` on an account that has children.

9.3 Account deletion & deactivation

- **System accounts** (`is_system_account=true`) cannot be deleted via the API.
- An account **with children** cannot be deleted; a **referenced** account cannot

be deleted (the reference check is a hook, extended when journal-line FKs arrive in 1C/1D). Otherwise a non-system, unused account may be hard-deleted.

- **Deactivation** (`is_active=false`) removes nothing — the account stays for

historical reporting and can be reactivated.

- **No monetary balance is stored on an account** (Rule 20); the ledger is the

single source of truth. No cached balances.

9.4 Accounting periods

- States in 1B: **OPEN** and **LOCKED** only (soft-close/reopen workflows deferred).
- `start_date <= end_date` (DB check + service).
- **Non-adjustment periods in one organization may not overlap.** A unique

constraint cannot express range non-overlap, so overlap is enforced at the **service layer inside the transaction** (portable). **Adjustment periods** (`is_adjustment_period=true`) are exempt and may overlap. Production PostgreSQL should additionally add a `btree_gist` **exclusion constraint** (see `DATABASE_SCHEMA.md`) — the residual TOCTOU race under high concurrency is the documented reason.

- **Lock** and **unlock** are **separate permissions**; **unlock is elevated**

(`accounting_periods.unlock`, held only by finance_manager/super_admin). A locked period is **never reopened silently**, and both lock and unlock are audited. A locked period cannot be edited or deleted.

- `row_version` provides optimistic-concurrency protection for lock/unlock.

> These rules prepare the ledger: Milestone 1D's posting engine will refuse to post into a LOCKED period and to a `allow_manual_posting=false` account.

9.5 Journal drafting & approval (Milestone 1C — implemented)

This milestone implements journal **drafting and approval only — no posting**.

- **Lifecycle (1C subset):** `DRAFT` → `APPROVED` or `DRAFT` → `REJECTED` → `DRAFT`.

Approval **only changes status** and has **no ledger effect**. The POSTED/REVERSED/VOIDED states and `PENDING_APPROVAL` in §2 are **not** part of 1C — they arrive with the posting engine in Milestone 1D.

- **A saved journal must always be valid:** ≥ 2 lines; every line one-sided and

non-negative (zero-value lines rejected); `SUM(debit) == SUM(credit)` exactly in `Decimal`; every line on an **active, manual-postable, same-organization** account. (This refines the Phase 0 "drafts may be unbalanced" note: because the API saves a whole entry atomically, balance is required on every save.)

- **Editing:** only a `DRAFT` may be edited; an `APPROVED` journal is read-only; a

`REJECTED` journal must be **returned to draft** before it can be edited again.

- **Numbering:** `journal_number` is unique per organization, gap-free, allocated

from `document_sequences` inside the create transaction (Rule 11); format `JE-000001`.

- **Segregation of duties:** the **creator may not approve their own journal**

(enforced; could later be made configurable per organization).

- **Locked periods:** a journal whose date falls in a **LOCKED** period (Milestone

1B) is rejected on create/update/approve. This is validation only — still no posting.

- **Permissions:** `journals.{read,create,update,approve,reject}` — approve/reject

are held by approver/finance_manager; create/update by accountant/bookkeeper.

- Every state change writes an append-only audit row.

9.6 Posting & reversal engine (Milestone 1D — implemented)

Posting turns an APPROVED journal into a POSTED ledger record. The posted journal lines themselves are the ledger at this stage — **no cached account balances are stored** (the ledger is recomputed from posted lines by the reports in 1E).

Sole posting path. `app/services/posting.py` is the only code that sets `POSTED`. No API, model hook, background process, test helper, or future AI service may bypass it. The reversal engine posts its reversal through the same function.

Transaction boundary. A post (or a full reversal) runs in **one** database transaction and either commits atomically or rolls back completely. Posting never alters debit/credit values. Validation re-checks, at posting time: status is APPROVED, ≥ 2 lines, balanced `Decimal` totals, every line one-sided, every account same-org + active + manual-postable, and the journal date is in exactly one **OPEN** period. A period **locked after approval** still blocks posting.

Immutability. After POSTED, the header, lines, amounts, date, accounts and status cannot be changed; a POSTED journal cannot be edited, approved, rejected or returned to draft (enforced in the service and API layers). Corrections are made only by a reversal. Row-level immutability is enforced in the application — a portable cross-database trigger is intentionally not used (see SQLite note below).

Idempotency. Posting is idempotent. An optional `idempotency_key` is stored on the journal; **UNIQUE** `(organization_id, idempotency_key)` prevents reuse for a different journal in the same organization (a conflicting reuse returns `409`). Re-posting an already-POSTED journal returns the existing result (a `journal_post_duplicate` audit event) and never applies the transaction twice. Different organizations may use the same key (tenant-scoped).

Concurrency. The target journal row is loaded `SELECT ... FOR UPDATE` (PostgreSQL row lock). The definitive guards are the **status transition** (an already-POSTED journal is never re-posted), the `row_version` optimistic-lock column (a concurrent update raises `StaleDataError`), and the **idempotency unique constraint**. **SQLite limitation (honest):** SQLite ignores `FOR UPDATE`; it serialises writers at the database-file level, so the state-based guards — not row locking — are what prevent a double-post in the SQLite dev/test environment. No distributed lock, Redis, or queue is used. The concurrency behaviour is verified by a **state-based** test ("repeated posting does not post twice"), not a threaded one.

Reversal workflow. Reversing a POSTED journal: validate it is POSTED, not already reversed, and not itself a reversal (reversal-of-reversal is rejected); allocate a **new journal number**; copy the original lines with **debit/credit swapped**, preserving account references and descriptions; add a "Reversal of <number>" description; link `reversal_of_journal_id` ↔ `reversed_by_journal_id`; post the reversal through the engine (its date must be in an OPEN period); then mark the original `REVERSED`. All in one transaction — if the reversal fails to post, neither journal is changed. The reversal is system-generated, so the creator-cannot-post segregation-of-duties check does not apply to it.

Segregation of duties. The journal **creator cannot approve** (1C) and **cannot post** their own journal. `journals.post` and `journals.reverse` are held by finance_manager and super_admin; approver may approve/reject but not post. Approver/poster separation beyond the creator rule is intentionally not enforced (kept simple, as permitted).

Source traceability. `source_type` (default `manual`, `reversal` for reversals) and `source_id` are generic — no invoice- or MyInvois-specific logic. Later document types will set these when they generate

journals.

10. Core ledger reports (Milestone 1E — implemented)

Two **read-only** reports: **Trial Balance** and **General Ledger** (`app/services/reports.py`). They never write and never use cached balances — all figures are computed from posted journal lines with `Decimal`.

10.1 Source of truth — "ledger-effective"

A journal is included iff it was **successfully posted**, identified by `` `posted_at IS NOT NULL` `` (not merely by current status == POSTED). This includes:

- **POSTED** journals;
- a **REVERSED original** — it was posted, so its `posted_at` remains set; it is

never dropped just because its status is now REVERSED (its history stands);

- its **POSTED reversal** — the opposite effect.

DRAFT / APPROVED / REJECTED journals have no `posted_at` and never appear. The original and its reversal therefore both show, and their opposite effects net to zero — the ledger keeps full history rather than erasing the original.

10.2 Date semantics

`from_date` and `to_date` are both **inclusive**. Reject `from_date > to_date` (`400`); an invalid date format is a `422`. Reports may span multiple periods, and **locked periods remain reportable**.

- **Opening** = ledger-effective activity strictly **before** `from_date`.
- **Period movement** = activity within `[from_date, to_date]`.
- **Closing** = opening + period movement.

10.3 Trial Balance

Per account: opening/period/closing **debit & credit** columns. Opening and closing are the account's net balance split onto the debit or credit side by sign; period debit/credit are the gross movements in the range. Because every posted journal balances, **total closing debit == total closing credit** (and total period debit == total period credit). Accounts with no activity are omitted unless `include_zero=true`; an **inactive** account that has activity stays visible.

10.4 General Ledger

Line-by-line posted ledger with a **per-account running balance**, deterministically ordered by `journal_date, journal_number, line sequence, line id`. The running balance follows the account's `normal_balance` (so contra accounts are handled automatically): for a debit-normal account a debit increases and a credit decreases the balance; for a credit-normal account the reverse. Each account's running balance is **seeded by its opening balance** (activity before `from_date`). A reversal appears as its **own separate transaction** and the original remains visible. Each row carries the journal number/description, line description, `source_type`/ `source_id`, posting date, and the reversal linkage.

10.5 Isolation

Organization context comes only from the authenticated principal — never a query parameter. Cross-organization data never appears; a General Ledger `account_id` from another organization is rejected (`404`). Read access is gated by `reports.trial_balance.read` / `reports.general_ledger.read`.

11. Sales quotations & invoices (Milestone 2B — implemented)

11.1 Quotation lifecycle (no ledger effect)

`DRAFT → APPROVED → CONVERTED`, plus `DRAFT → REJECTED → DRAFT`, and `DRAFT/APPROVED/REJECTED → CANCELLED`. Only a **DRAFT** may be edited; APPROVED, CANCELLED and CONVERTED are read-only. Only an **APPROVED** quotation may convert, and only **once**. Conversion is atomic: it creates a new **DRAFT** invoice (own `INV-` number), copies the customer and lines, recalculates totals, sets the quotation `CONVERTED`, and links both sides (`converted_invoice_id ↔ source_quotation_id`). Quotations never touch the ledger.

11.2 Invoice lifecycle

`DRAFT → APPROVED`, and `DRAFT/APPROVED → CANCELLED`. Only a **DRAFT** may be edited; APPROVED and CANCELLED are read-only. Invoice numbers are unique per org.

11.3 Totals

Computed on the server, in `Decimal`, and recalculated on every create/update (and again at approval). Per line: `gross = quantity × unit_price`, `line_total = gross − discount + tax` (for tax-INCLUSIVE lines the tax is already inside `gross − discount`, so `line_total = gross − discount`). Document: `subtotal = Σ gross`, `discount_total = Σ discount`, `tax_total = Σ tax`, `total = Σ line_total` (equal to `subtotal − discount_total + tax_total` for EXCLUSIVE/legacy lines; summing line totals is what keeps tax-INCLUSIVE lines from being double-counted — see §16). Rules: `quantity > 0`, `unit_price ≥ 0`, `discount ≤ gross`, `tax ≥ 0`, `line_total ≥ 0`, at least one line. Client-supplied totals are never trusted.

11.4 Invoice → journal mapping (on approval)

Approving a DRAFT invoice creates a **DRAFT** journal through the existing journal service (`app/services/journals.build_draft`), atomically in one transaction, and **never posts it**:

- **Dr Accounts Receivable** — the invoice `total`.
- **Cr revenue account(s)** — each line's net revenue (`gross − discount`),

aggregated per revenue account.

- **Cr tax-payable** — `tax_total`, only when `tax_total > 0`.

The journal is balanced (`Dr AR = Σ revenue net + tax = total`). Each revenue account is validated (belongs to the org, type `REVENUE`, active, manual-posting). The invoice is linked via `journal_entry_id`; the journal records `source_type='sales_invoice'`, `source_id=<invoice id>`. Approval fails (and rolls back) if the total is not > 0, a revenue account is invalid, or a required system account is missing.

11.5 Required system accounts

Resolved by **stable** `system_code`, never by name or hard-coded id:

- `accounts_receivable` — required for every invoice approval. If absent,

approval fails with a clear message to install/configure the chart of accounts (the account is never created silently). The Malaysian starter chart provides it.

- `sst_payable` — required only when `tax_total > 0`; if absent, a taxed

approval fails clearly. **The starter chart does not flag a tax-payable system account**, so taxed invoices require one to be configured first. *(TODO: a future milestone should let an organisation designate its tax-payable account; there is no API to set* `system_code` *in 2B.)* Tax is entered explicitly per line — there is **no SST calculation, no tax-rate table, and no LHDN connection**, and running successfully is not a claim of tax compliance.

11.6 Separation of duties & the invoice journal

The DRAFT journal is `created_by` the **invoice approver**. By the journal's existing SoD rule (a creator cannot approve or post their own journal), a **different** user must approve and post it — so posting an invoice's revenue to the ledger always involves at least two people. Invoice approval never posts.

11.7 Cancellation limits

- A **DRAFT** invoice cancels freely (no journal).
- An **APPROVED** invoice whose linked journal is **not posted** cancels: the

non-posted journal has no ledger effect and is **neutralised** (set `REJECTED`, atomically) so it can never be posted for a cancelled invoice.

- If the linked journal is **POSTED or REVERSED**, cancellation is **rejected** —

a **credit note** (a future milestone) is required. Posted invoices are never reversed automatically, and invoices are never deleted.

---

12. Supplier bills (Milestone 2C — implemented)

12.1 Lifecycle

`DRAFT` → `APPROVED`, and `DRAFT/APPROVED` → `CANCELLED`. Only a **DRAFT** may be edited; APPROVED and CANCELLED are read-only. Only a DRAFT may be approved. Bill numbers (`BILL-`) are unique per org and server-allocated. Approval creates a DRAFT journal and never posts it. Bills are never deleted.

12.2 Totals

Identical to sales (server-computed `Decimal`, recalculated on every create/update and at approval): per line `gross = quantity × unit_price`, `line_total = gross − discount + tax`; document `subtotal = Σ gross`, `discount_total = Σ discount`, `tax_total = Σ tax`, `total = subtotal − discount_total + tax_total`. `quantity > 0`, `unit_price ≥ 0`, `discount ≤ gross`, `tax ≥ 0`, `line_total ≥ 0`, ≥ 1 line.

12.3 Line accounts

Each line references an **EXPENSE** account, or an **ASSET** account when a purchase is intentionally capitalised. Validated (org, allowed type, active, manual-posting); a REVENUE account is rejected. Accounts are never auto-classified (no AI). Fixed-asset registers / depreciation are out of scope.

12.4 Bill → journal mapping (on approval)

Via `journals.build_draft` (atomic, never posted):

- **Dr expense/asset account(s)** — each line's net (`gross − discount`), aggregated

per account.

- **Dr tax-recoverable** — `tax_total`, only when `> 0`.

- **Cr Accounts Payable** — the bill `total`.

Balanced (`Cr AP = Σ net + tax = total`). `source_type='supplier_bill'`, `source_id=<bill id>`; the bill is linked via `journal_entry_id`.

12.5 Required system accounts

Resolved by **stable** `system_code`, never by name/id:

- `accounts_payable` — required for every bill approval; if absent, approval

fails with a clear message to install/configure the chart of accounts (the starter chart provides it; it is never created silently).

- `sst_claimable` (tax-recoverable / input tax) — required only when

`tax_total > 0`; if absent, a taxed approval fails clearly. The starter chart does **not** flag one, so recoverable-tax bills need one configured first. *(TODO: a future milestone should let an org designate its tax-recoverable account; no API sets* `system_code` *yet.)* Tax is entered explicitly per line — no SST calculation, no assumption that the business can recover SST, and no LHDN connection. Success is not a claim of tax compliance.

12.6 Separation of duties

The DRAFT journal is `created_by` the **bill approver**, so by the journal's SoD rule a **different** user must approve and post it.

12.7 Duplicate supplier invoice

Within one org + supplier, a non-empty `supplier_invoice_number` cannot repeat (trimmed, case-insensitive). Blank/NULL always allowed; different suppliers / organisations may reuse a number. A rejected duplicate is audited. No fuzzy matching, no AI.

12.8 Cancellation limits

- A **DRAFT** bill cancels freely (no journal).
- An **APPROVED** bill whose linked journal is **not posted** cancels: the

non-posted journal is **neutralised** (set `REJECTED`, atomically).

- If the linked journal is **POSTED or REVERSED**, cancellation is **rejected** — a

debit note / adjustment (a future milestone) is required. Posted journals are never reversed automatically; debit notes are not implemented in 2C.

---

13. Customer & supplier payments (Milestone 2D — implemented)

13.1 Lifecycle

`DRAFT` → `APPROVED`, and `DRAFT/APPROVED` → `CANCELLED`. Only a DRAFT may be edited; APPROVED and CANCELLED are read-only (amounts and allocations frozen once approved). Numbers (`CP-/SP-`) are unique per org and server-allocated. A payment never posts to the ledger — approval creates a **DRAFT** journal (never posted). Payments are never deleted.

13.2 Allocation rules

A payment must allocate to at least one document, and the **allocations must add up to exactly the payment amount** (no unallocated amount, no overpayment). Each allocation amount > 0; a document cannot appear twice in one payment; the document must belong to the org and the selected party, be **APPROVED** (not draft/cancelled), match the payment currency, and the allocation must not exceed the document's current `amount_due`. All computed and validated on the server with `Decimal`.

13.3 Partial payments & concurrency

Partial payments are supported via the invoice/bill `amount_paid` / `amount_due` fields (document status stays APPROVED — no PAID/PARTIALLY_PAID). At **approval** the allocated documents are loaded `SELECT ... FOR UPDATE` and re-validated against their **latest** `amount_due`, so two drafts that each allocate the full outstanding can never both approve — the second fails. Invariants hold: `amount_paid` and `amount_due` never go negative, and `amount_paid + amount_due` always equals the document total. **SQLite limitation (honest):** SQLite ignores `FOR UPDATE` and serialises writers at the file level; the definitive guards are the amount_due re-check inside the approval transaction and `row_version`. PostgreSQL adds true row locking. No Redis / distributed locks. Concurrency is verified state-based (not with real threads), so no threaded test is claimed.

13.4 Journal mapping (on approval, atomic)

Via `journals.build_draft` (never posted):

- **Customer payment:** Dr **deposit account** (an ASSET, e.g. bank/cash) / Cr

Accounts Receivable — the payment amount.

- **Supplier payment:** Dr **Accounts Payable** / Cr **payment account** (ASSET) —

the payment amount. AR/AP are resolved by `system_code` (`accounts_receivable` / `accounts_payable`; approval fails clearly if missing). The deposit/payment account must be an active, manual-posting **ASSET** account, chosen by the user (never guessed). Each allocated document's `amount_paid += alloc` and `amount_due -= alloc`. `source_type` is `customer_payment` / `supplier_payment`, `source_id` the payment id; the payment is linked via `journal_entry_id`. **SoD:** the approver is the journal's `created_by`, so a different user must post it.

13.5 Cancellation

- A **DRAFT** payment cancels freely (no accounting effect).
- An **APPROVED** payment whose linked journal is **not posted** cancels atomically:

the allocated documents' balances are **restored** (`amount_paid -= alloc`, `amount_due += alloc`) and the non-posted journal is neutralised (→ REJECTED).

- If the linked journal is **POSTED or REVERSED**, cancellation is **rejected** — a

future refund, credit note or debit note is required. Posted payment journals are never reversed automatically.

13.6 Reference policy

The external `reference` (bank reference) is **not** unique — different bank transactions may share one. The internal payment number is the controlled unique id. References are trimmed. No fuzzy duplicate detection, no AI.

---

14. Receivables & payables ageing (Milestone 2E — implemented)

Two read-only reports that age unpaid documents into overdue buckets. They add **no** new balance system and **no** cached/snapshot ageing table — every figure is derived on demand from data that already exists.

14.1 Source of truth

- **Receivables** ages **APPROVED sales invoices**; **payables** ages **APPROVED

supplier bills**. DRAFT and CANCELLED documents are excluded, as are documents whose outstanding balance (as of the report date) is `<= 0`.

- A document's outstanding is its **original total** minus the sum of its

approved payment allocations. Ageing is computed from the document/payment data — **never** from journal balances, and never from a cached figure.

14.2 Historical `as_of_date` reconstruction

`as_of_date` (default: today) is **inclusive**. A report is a faithful snapshot of that past date, not of today:

1. Only documents dated on or before `as_of_date` are included.
2. Outstanding = original total − Σ(approved allocation amounts whose **payment

date ≤ `as_of_date`**).

1. A payment dated **after** `as_of_date` does **not** reduce the historical

outstanding — so the current `amount_due` (which reflects today's payments) is deliberately **not** used for a historical report.

1. Only **APPROVED** payments count. DRAFT payment allocations are ignored, and a

CANCELLED payment stops reducing the outstanding (its parent payment is no longer APPROVED). Payment state is read from the payment record, never inferred from journals.

Worked example — invoice RM 1,000 due 10 Jun; RM 400 paid 20 Jun; RM 600 paid 20 Jul. As of **30 Jun**: paid 400, outstanding **600**. As of **31 Jul**: paid 1,000, outstanding 0 → the invoice drops off the report.

14.3 Bucket boundaries (fixed; not configurable this milestone)

`days_overdue = as_of_date − effective_due_date` in **whole calendar days**, where `effective_due_date` is the document's due date (falling back to the document date when no due date was captured).

Bucket	Rule
Current	`days_overdue <= 0` (due date on or after `as_of_date`)
1–30	`1 ≤ days_overdue ≤ 30`
31–60	`31 ≤ days_overdue ≤ 60`
61–90	`61 ≤ days_overdue ≤ 90`
More than 90	`days_overdue ≥ 91`

Every outstanding document falls in **exactly one** bucket. Boundary check: due = as-of → Current; 1 day over → 1–30; 30 → 1–30; 31 → 31–60; 60 → 31–60; 61 → 61–90; 90 → 61–90; 91 → More than 90.

14.4 Partial payments & integrity

Partial payments show only the **remaining** balance; fully paid documents are excluded. Money is `Decimal` throughout. Invariants: a customer/supplier total equals the sum of its documents, the grand total equals the sum of all parties, and each detail row's `amount_paid + amount_outstanding == original_total` exactly. Outstanding is never negative (rows at/below zero are dropped). Inactive customers/suppliers with an outstanding balance remain visible.

14.5 Isolation & read-only guarantee

The organisation is always taken from the authenticated principal — a query never carries a trusted organisation id. A `customer_id`/`supplier_id` filter must resolve within that organisation (else `404`). The service never writes and never modifies invoice, bill, payment or journal records; repeated identical requests return identical output.

14.6 Out of scope (deferred)

Statements, reminders/collections, credit/debit notes, refunds, write-offs, bad-debt/expected-credit-loss provisions, cash-flow forecasting, tax reports, bank reconciliation, AI/OCR, document upload, MyInvois, and PDF/Excel/CSV export are **not** part of this milestone.

---

15. Cash/card expenses (Milestone 3C — implemented)

An **expense** records money paid directly (from an ASSET account such as Cash on Hand or a bank account) rather than through Accounts Payable. It can be entered manually or created from an AI-**reviewed RECEIPT**.

15.1 Lifecycle & eligibility

`DRAFT` → `APPROVED`, and `DRAFT`/`APPROVED` → `CANCELLED`. Only a DRAFT is editable; APPROVED/CANCELLED are read-only. Numbers (`EXP-`) are org-unique and server-allocated. An expense may be created **from a document only when it is a** `RECEIPT` **with status** `REVIEWED` — supplier invoices (which create a supplier bill), and UPLOADED/EXTRACTED-unreviewed/REJECTED/FAILED documents, are refused. **One reviewed receipt creates at most one expense** (DB-backed by `UNIQUE(source_document_id)` + the `created_expense_id` back-link).

15.2 Suggestions are suggestions (never automatic)

Supplier matching (TIN → registration → exact name → case-insensitive name) is optional and only *pre-selects*; a receipt needs no supplier and no supplier is auto-created. Expense-account suggestions rank existing active, manual-posting EXPENSE/ASSET accounts with a confidence; the user chooses the account for **every** line. No account is created or modified. The **payment account** (an active, manual-posting **ASSET**) is chosen by the user — never inferred.

15.3 Totals

Per line: `gross = quantity × unit_price`, `line_total = gross − discount + tax` (`quantity > 0`, `unit_price ≥ 0`, `discount ≤ gross`, `tax ≥ 0`, `line_total ≥ 0`). Expense: `subtotal = Σ gross`, `discount_total = Σ discount`, `tax_total = Σ tax`, `total = subtotal − discount_total + tax_total`. At least one line is required. Totals are always recomputed server-side in `Decimal`; browser totals are never trusted. Currency defaults to MYR; no FX conversion.

15.4 Tax treatment (the user's choice — never AI/tax-law)

When `tax_total > 0` the user chooses one of two simple treatments (AI only extracts the number; it never decides recoverability, SST, deductibility, or a tax code):

- **include** — each line's tax is folded into its expense/asset debit;
- **recoverable** — the total tax is debited to the configured `sst_claimable`

account (approval fails clearly if that account is not configured).

15.5 Journal on approval (DRAFT only, atomic)

Via `journals.build_draft` (never posted): **Dr** expense/asset account(s) (aggregated per account; net-of-tax when *recoverable*, tax-inclusive when *include*) **[+ Dr** `sst_claimable` **= tax_total** when *recoverable*] / **Cr** the payment account (= `total`). `source_type='expense'`, `source_id=<expense id>`. The approver becomes the journal's `created_by`, so a **different** user must post it (SoD). Cached account balances are never updated.

15.6 Cancellation

A DRAFT expense cancels freely. An APPROVED expense whose linked journal is **not posted** cancels atomically (the non-posted journal is neutralised → REJECTED). If the journal is **POSTED or REVERSED**, cancellation is **rejected** — a future reversal/adjustment is required. Expenses are never deleted.

15.7 Out of scope (deferred)

Employee expense claims, staff reimbursements, mileage claims, corporate cards/card feeds, bank reconciliation, payment matching, automatic supplier/account creation, automatic approval/posting, tax advice, and MyInvois are **not** part of this milestone.

16. Tax configuration & calculation (Milestone 4A — implemented)

Configurable **SST** (Sales & Service Tax) codes and one shared, deterministic tax engine that sales invoices, supplier bills and expenses all call. This is *accounting* tax calculation and configuration only — it makes **no** tax-compliance or e-Invoice/MyInvois claim, submits nothing to LHDN or MyInvois, and gives no tax advice. Three concerns are kept separate: (1) accounting tax calculation (this section); (2) SST configuration (codes + org settings); (3) a *future* MyInvois classification (the `external_classification` field, unused for calculation).

16.1 The one shared engine

`app/services/tax_engine.py` is pure: `Decimal` **only, no DB, no AI, no** `float`. `rate` is a percentage (6 = 6 %), applied as `rate ÷ 100`. Rounding is commercial **ROUND_HALF_UP** at the organisation's configured precision (default 2). Calculation is **line-level, then document summation** (never document-level tax):

- **EXCLUSIVE** — `net = round(gross − discount)`; `tax = round(net × rate)`;

`total = net + tax`.

- **INCLUSIVE** — `gross − discount` already includes tax; `net = round(total ÷ (1 +

rate))`; tax = total − net; total = gross − discount`.

- **ZERO / EXEMPT / OUT_OF_SCOPE** (or a zero rate) — `tax = 0`; `net = total = gross −

discount`.

Sales, bills and expenses must **not** each hold their own tax formula — they all call `tax.compute_line_amounts()`, which delegates to this engine.

16.2 Codes, directions and account mapping

A **tax code** has a direction (`SALES` / `PURCHASE` / `BOTH`), a mode, a rate, and an optional mapped account. Validation (never silently creating accounts):

- **Sales** tax with a rate → must map to an **active, manual-posting LIABILITY** account.
- **Recoverable purchase** tax → maps to an **active, manual-posting ASSET** account.
- **Non-recoverable purchase** tax → **no** account (the tax is folded into the

expense/asset cost).

- Zero-family codes carry no account.
- Cross-org, inactive, or wrong-type accounts are **rejected**.

16.3 Journal mapping (coded lines)

On approval the shared helpers post a coded line's tax to its **mapped** account:

- **Sales** — `Cr` the mapped liability, per account (revenue is credited at **net**;

`Dr AR = grand total`).

- **Recoverable purchase / expense** — `Dr` the mapped asset (expense/asset debited at

net; `Cr AP` or the payment account = **grand total**).

- **Non-recoverable purchase / expense** — the tax is **added to the expense/asset

debit**; no separate tax line.

Legacy lines (an explicit `tax_amount` with no `tax_code_id`) keep the pre-4A behaviour, routing tax to the system `sst_payable` (sales) / `sst_claimable` (purchase/expense) accounts resolved by `system_code`. For expenses, the existing `tax_treatment` (include / recoverable) still governs legacy lines.

16.4 Historical integrity & effective dates

Each transactional line preserves `tax_code_id`, `tax_rate`, `tax_mode`, `net_amount`, `tax_amount` and `line_total` **as calculated at the time**. A **draft** uses the code's **current** rate/mode and snapshots them onto the line; at **approval** the stored snapshot is used, so totals never depend on the code's *current* rate. Changing a tax code later affects only **future draft** calculations — approved documents and posted journals are immutable and are **never** recalculated or bulk-migrated.

Effective dates. A tax code's optional `effective_from`/`effective_to` window is enforced **only when a new draft line resolves the current code**: if the document's date falls outside the window, the code cannot be applied (clear 400). Because approval and any later recalculation read the line's **snapshot** (never re-resolving the code), narrowing or expiring a code's window afterwards **cannot** change or block an already-snapshotted/approved document. Converting an approved quotation to an invoice **carries the snapshot forward** (code id + rate + mode + net), so the code is neither lost nor re-resolved on conversion.

16.5 AI boundary

AI never chooses a tax code, decides recoverability, decides SST registration, or decides any legal tax treatment — the user chooses and confirms. If a calculated tax differs materially from an AI-extracted amount, the difference is surfaced without silently overwriting, and the extracted value is preserved in extraction history. See `docs/AI_GOVERNANCE.md`.

17. Bank accounts & reconciliation (Milestone 4C — implemented)

Deterministic — **no AI, no float**. Everything reconciles to the ledger.

17.1 Bank accounts

Each bank account maps **1:1 to an existing ASSET ledger account** — the ledger account is **never auto-created**, must be active and cannot be shared by two bank accounts.

17.2 Statement import

`Upload → Parse → Preview → Validate → Import → Match → Reconcile`. Nothing is imported automatically — the user confirms after previewing. Parsing sits behind **one parser interface** (`CSV` with configurable column mapping, `OFX` bank `STMTTRN`, `CAMT.053` foundation). A stable per-line hash (account + date + amount + reference + description) dedupes a re-imported statement so it never creates duplicates.

17.3 Matching (suggest only)

The engine proposes candidate links to **existing** ledger records in priority order: exact payment reference · exact amount + date · exact invoice number · exact supplier-bill number · exact journal reference · a configured bank rule — each with HIGH/MEDIUM/LOW confidence. It **never** auto-applies, **never** posts and **never** auto-creates journals; the user confirms. Matching creates a **link** only and never modifies the linked record. A record already matched is never re-suggested.

17.4 Splits

One bank transaction may be matched to **several** records via multiple `bank_matches` whose amounts sum **exactly** to the transaction amount (validated in `Decimal`; no rounding error). Each part's sign must match the bank line.

17.5 Bank rules

A rule *suggests* a category / ledger account / tax code / description for an unmatched transaction (text operator `CONTAINS`/`STARTS_WITH`/`ENDS_WITH`/`REGEX` + optional amount range, by priority). Rules are **never executed automatically**.

17.6 Reconciliation math

For a bank account:

- `book_balance = opening_balance + Σ(debit − credit)` over **posted** journal lines on the

bank's ledger account.

- A ledger item is **cleared** once its journal is linked to a bank transaction (directly,

or via its payment / expense).

- `outstanding_deposits = Σ` uncleared **debits** (money in, in the books, not yet on the

statement); `outstanding_payments = Σ` uncleared **credits** (money out, not yet cleared).

- `expected_statement = book_balance − outstanding_deposits + outstanding_payments`; the

difference vs the actual statement balance is `0` when reconciled. A health indicator (Green balanced / Yellow minor / Red reconciliation required) is shown — **no accounting advice**.

17.7 Out of scope (deferred)

Open Banking / bank APIs / direct feeds, ML/AI matching, receipt/expense OCR, email/WhatsApp import, and multi-currency conversion of statement lines are **not** in this milestone.

18. Fixed assets & depreciation (Milestone 4D — implemented)

Deterministic — **no float**. Users never calculate depreciation by hand.

18.1 Categories & assets

A **category** carries the default depreciation method, useful life (months), residual % and the ledger accounts (asset, accumulated depreciation, expense, disposal gain/loss). A **fixed asset** inherits those (overridable) and maps to an **ASSET** asset account, an **ASSET** (contra) accumulated-depreciation account and an **EXPENSE** depreciation account — all validated, never auto-created. Cost / method / life / residual are **locked** once depreciation has started.

18.2 Depreciation methods

- **STRAIGHT_LINE:** monthly = `(cost − residual) / useful_life_months`.
- **DECLINING_BALANCE:** monthly = `opening_book_value × (annual_rate/100) ÷ 12`.
- **UNITS_OF_PRODUCTION: framework/foundation only** — needs per-period production units

this milestone does not collect, so it depreciates 0 (documented). Every method **respects the residual value**: the book value never falls below residual, and each period is capped so it cannot overshoot the remaining depreciable base.

18.3 Engine & workflow

The engine produces `DepreciationEntry` rows **DRAFT** → **APPROVED** → **JOURNALED**. Generating the journal uses the existing journal service (never bypassing the posting engine): one balanced DRAFT entry per depreciation, `Dr depreciation expense / Cr accumulated depreciation`. The asset's accumulated depreciation + book value update on journaling; the journal is **posted separately** (no auto-posting). Running a month is idempotent (one entry per asset per month).

18.4 Disposal & write-off (journal mapping)

On disposal the engine computes cost, accumulated depreciation, book value and gain/loss and builds a balanced DRAFT journal: `Dr accumulated depreciation` (accumulated) · `Dr proceeds account` (sale proceeds) · `Cr asset account` (cost) · then `Cr disposal gain` (proceeds – book value, if positive) or `Dr disposal loss` (book value – proceeds, if negative). A **write-off** is a disposal with no proceeds (`Dr accumulated + Dr loss / Cr asset`). Disposed assets are **never deleted**; they stay in the register with their movement record.

18.5 Transfer & impairment

- **Transfer:** changes location / department / responsible person — **no ledger effect**,

audit only.

- **Impairment (foundation, no IAS 36):** records amount + reason + date and a DRAFT journal

`Dr loss / Cr accumulated depreciation`, capped so book value never goes below zero.

18.6 Out of scope (deferred)

Maintenance scheduling, asset-tracking hardware, barcode/QR labels, IoT, GIS, fleet, insurance, tax filing, and full IAS 36 impairment modelling are **not** in this milestone.

19. AI CFO — read-only analysis (Milestone 4E)

The AI CFO adds **no** posting rules. It is a **read-only** intelligence layer: it never creates, edits, posts, approves, reconciles or deletes anything, and it duplicates **no** accounting logic. All figures are derived from the posted ledger via `reports.trial_balance` and the existing ageing / bank / fixed-asset / tax services; the AI only narrates over those deterministic numbers and never produces or changes a figure. Insights reference actual computed values; recommendations are **operational** only (never accounting, tax, legal or investment advice). See `docs/AI_GOVERNANCE.md` §9.

20. Forecasting, budgeting & scenarios — isolated from the ledger (Milestone 4G)

The planning module adds **no** posting rules and is **isolated from the ledger**. Actual accounting remains the single source of truth.

- **Forecasts and scenarios are read-only projections; budgets are editable planning numbers.**

None of them creates, edits, approves, posts, reverses or modifies a journal, and the module has no access to any posting/approval code path. A test asserts that generating a forecast or scenario creates **zero** journal entries.

- **Budgets** live in their own tables (`budgets`, `budget_lines`) with a Draft → Approved →

Archived lifecycle and multiple versions per fiscal year, of which **at most one is active**. Approving/archiving changes planning state only.

- **"Actuals" for budget-vs-actual** are read from the posted ledger via `reports.trial_balance`

(revenue = period credit − debit; expense = period debit − credit) — never recomputed and never written back.

- **Every projected number is deterministic** (`app/services/forecast/`), computed by reusing

the AI CFO baseline; the AI only narrates and never produces a figure. Each forecast stream carries a formula, inputs, assumptions, a deterministic confidence score, its data period and its source. The tax estimate is a labelled planning figure, **not a tax return, filing or advice**; nothing here is tax, legal or investment advice. See `docs/AI_GOVERNANCE.md` §11.