Technical Governance Framework (TGF) v1.8.1

Et al Solutions LLC

Date: February 27, 2026
Author: David Omer (Founder)

1) Purpose

The Technical Governance Framework (TGF) defines Et al Solutions LLC's complete IT/R&D way of working. It is not an index—it is policy. TGF establishes the standards, principles, and behavioral expectations that govern all technical activity. This policy is established in order to:

- Make behavior predictable across repos and clients.
- Minimize accidental divergence between environments.
- Support rapid onboarding and handoff.

2) Customer Context Consideration

All delivery plans must account for:

- **Location:** favor local or regionally stable providers.
- **Technical Profile:** assume non-technical users prefer no-maintenance solutions.
- **Simplicity Bias:** default to managed/static platforms unless otherwise requested.

3) Tooling Discipline (new in v1.5)

All personnel, contractors, and A.I. agents must use tools stored in the *ops* directory of the **ops and sops** repository whenever an appropriate tool exists for the task. These tools form the approved enforcement and quality layer for EPC and TGF compliance. Alternate tools or manual approaches may only be used when:

1. No existing tool covers the scope, or
2. A temporary exception is documented and approved. This ensures consistency, traceability, and reliability across all operations.

4) Problem-Solving Discipline (new in v1.5)

In **production or live systems**, triage is mandatory: restore service immediately, then investigate root cause after stability is restored. In **development, tooling, research, or new business lines**, root-cause resolution takes priority. Teams must make at least **two to three good-faith attempts** to correct the underlying issue before implementing alternate paths or workarounds. Workarounds are acceptable only after documented root-cause efforts are completed or time-boxed.

5) Communications & Documentation Standards

- Use clear, human-centered language.
- Maintain transparency about methods and results.
- Keep tone consistent with brand identity.
- Include version/date on all official documents.
- Prefer Markdown or plain text.

6) Enforcement

Compliance with TGF and EPC is mandatory. Exceptions require documented approval.

7) Repositories & Branching

7.1) Repository Naming

- Internal project repos follow the pattern:

  ```
  ##SOURCE-INITIAL: f=Freelancer, u=Upwork, etc.
  [SOURCE-INITIAL]-[PID] [Descriptive Name]
  ```

  Example (sourced from Freelancer): `f-39797271 MVP Task Manager`

- Public-facing repo names MAY be customer- or product-branded as required, but internal references SHOULD still use the `[SOURCE-INITIAL]-[PID]` prefix.

7.2) Default Branch

- Default branch is `main`.
- `main` MUST always be deployable.
- Long-running feature branches SHOULD be avoided; prefer short-lived feature branches rebased onto `main`.

7.3) Branch Types

- `main` – Production-ready code.
- `feat/*` – New features.
- `fix/*` – Bug fixes.
- `chore/*` – Tooling, docs, refactors without behavior change.
- `spike/*` – Throwaway exploratory work (MUST NOT be merged to `main` without cleanup).

8) Coding Standards

8.1) General

- Favor clarity over cleverness.
- No silent failure: scripts MUST `set -euo pipefail` or equivalent.
- All external tools MUST be invoked with explicit options (no reliance on implicit defaults).

8.2) Shell

- All project scripts MUST be POSIX shell or Bash with:

```
#!/usr/bin/env bash
set -euo pipefail
```

- Functions MUST be idempotent where reasonable and explicitly documented when they are not.
- Use long options for readability (`--project` , `--region` , etc.) where available.
- Functions that return values MUST use **safe-return semantics**: accept an output variable name from the caller and assign into it via `printf -v`. Functions MUST NOT rely on stdout to return values.
- Functions that are consumed by other scripts MUST send all human-readable logs to stderr (`>&2`). Stdout is reserved exclusively for machine-readable data (for example: JSON, digests, platform lists).
- Command substitution (`var="$(func)"`) MUST NOT be used with functions that emit logs. Only pure data-emitting CLI tools may be used in command substitution.

8.2.1) Control-flow + Logging Primitives (sys-/ops- standard)

All shell-based tooling intended to be sourced (helper libraries) or invoked by other scripts MUST use a unified control-flow contract that cleanly distinguishes "exit this layer" vs. "red button" behavior.

Control primitives

- `bail <rc>` :
  - Purpose: early exit from the current layer (success or failure) without "red button" behavior.
  - Required semantics:
    - If the current file was sourced: MUST `return <rc>` .
    - If the current file was executed: MUST `exit <rc>` .
- `fatal <rc>` :
  - Purpose: stop-the-world conditions.
  - Required semantics: MUST always `exit <rc>` (even when sourced).

Logging + exit helpers

- `ok "msg"`:
  - Meaning: “we are fine; we’re done early; caller may continue.”
  - Behavior: log + `bail 0`.
- `warn "msg"`:
  - Meaning: “probably fine; investigate later.”
  - Behavior: warning log; continue.
- `die "msg" [rc]`:
  - Meaning: “this layer hit a critical problem; caller may or may not consider it fatal.”
  - Behavior: error log + `bail rc` (default `rc=1`).
- `fatal_error "msg" [rc]`:
  - Meaning: “red button; stop everything now.”
  - Behavior: fatal log + `fatal rc` (default `rc=1`).

Channel discipline

- `warn`, `die`, and MUST log to stderr.

`fatal_error`

- Scripts/functions whose stdout is consumed by other tooling MUST keep human-readable logs on stderr; stdout remains reserved for machine-readable data (JSON, digests, filenames, etc.).
- Top-level, human-invoked commands MAY print human-readable output to stdout when they are not intended to be piped.

8.3) JavaScript / TypeScript

- Use modern syntax (ES modules, async/await) unless the runtime forbids it.
- Linting with ESLint and a consistent style (Prettier or equivalent) is strongly recommended.

8.4) PHP / WordPress

- Follow WordPress coding standards for spacing, naming, and file layout where applicable.
- Custom PHP logic SHOULD be wrapped in clear functions or classes rather than placed inline in templates.

9) Configuration & Secrets

- Configuration is provided via environment variables only.
- Secrets MUST NOT be committed to version control.
- `.env` files MAY be used directly in **development** only
- `.env` files MAY be used indirectly in **all other** to prime the environments native environment variable mechanism
- Production runtimes MUST read configuration from:
  - Platform-level environment variables.
  - Secret Manager / Key Vault / Parameter Store, and/or

10) Testing & CI Policy

10.1) General Principles

10.1.1) Every project MUST have at least a **smoke test** that can be run in CI.
10.1.2) For new work, tests SHOULD be written alongside the feature (or immediately after) rather than deferred.
10.1.3) CI MUST run on every push to `main` and every PR targeting `main`.

10.2) Stack Matrix

10.2.1) JavaScript / TypeScript (Web, Node, React)

- **Unit/Integration tests**: Vitest (preferred) or Jest.
- **React component tests**: React Testing Library.
- **E2E tests** (where justified): Playwright.

Minimal:

- At least one test suite executed by CI.
- Basic smoke test for core flows.

Preferred:

- Coverage for critical paths (auth, payments, data mutation).
- E2E for high-risk flows (checkout, destructive admin actions).

10.2.2) PHP / WordPress

- **Unit tests**: PHPUnit.
- **Integration tests**: where practical, use a test database schema and automated setup.
- **E2E**: optional; may use Playwright or Cypress against a running container.

Minimal:

- A smoke test verifying that:
  - WordPress boots.
  - Core routing works.
  - Health endpoint is responsive (if defined).

10.3) Infrastructure / Scripts

- Idempotent scripts (init, deploy, configure) MUST have a test mode or be structured so they can be executed safely in a non-prod environment.
- Infrastructure-as-code (Terraform, etc.) SHOULD have validation steps (formatting, plan checks) in CI.

10.4) CI Configuration

- CI SHOULD be defined in the repo (`.github/workflows` , `.gitlab-ci.yml` , etc.).
- Minimal pipeline:
    1. Check out code.
    2. Install dependencies.
    3. Run tests (unit + integration where applicable).
    4. Optionally build a preview artifact (container, bundle).
- For containerized stacks, CI SHOULD build the container and run tests inside it, or build once and reuse the image in subsequent steps.

10.5) Quality Gates

- PRs to `main` MUST be blocked if:
    - Tests fail.
    - Linting fails (if configured).
- Code reviews SHOULD confirm:
    - Tests exist for new behavior.
    - Risky changes have appropriate coverage.

11) Environment & Deployment Standards

Maintain predictable, secure, reproducible environments following EPC.

Key ideas:

1. **Identity**: Engagements are tracked via `PROJECT_ID` ; environment instances are named via `STACK` ; individual releases are identified via `DEPLOYMENT_ID` ; cloud providers are referenced via `CLOUD_PROJECT_ID` .
2. **Configuration**: All runtime configuration flows from environment variables.
3. **Portability**: The same `.env*` model drives local and cloud adapters without editing code.
4. **Safety**: Production never consumes `.env` files directly.

11.1) Environment Identities

11.1.1) APP_SLUG

APP_SLUG is the primary identifier of a project and it can be replicated for different instances, if necessary.

- Assigned once per engagement (e.g., sourced from Freelancer, Upwork, or an internal quoting system).
- Stable across all environments (DEV, QAS, PRD, etc.) and runtimes.
- Used for billing, reporting, and governance — never overloaded to mean a cloud provider project or a container deployment.

Typical examples:

- f-39952777-sharepoint-showcase
- f-40197132-apify

Constraints (governance-level, not enforced by any specific cloud):

- 2–32 characters.
- lowercase a-z, digits, and hyphens (`-`) only.
- SHOULD start with a letter designating the source (e.g. f, u, e, etc) followed by the source project id/number and a brief description

Recommended regex: `^[a-z][a-z0-9-]{1,31}$`

Once chosen, APP_SLUG MUST remain stable for the life of the engagement.

11.1.2) PROJECT_ID (Et al Canonical Project / Engagement ID)

PROJECT_ID is the Et al canonical identifier for a customer engagement or job.

Format & Constraints:

- 6–30 characters
- [a-z0-9-] only
- Must start with [a-z]
- Must end with [0-9]

NOTE: the .project_timestamp was automatically written to the repository folder by the sys-clone-project tool and sys-setup* scripts call a utility which automatically formats PROJECT_ID and writes back to env file

Example:

- APP_SLUG = "patriotic-F39706271"
- .project_timestamp = ”1764804660”
- PROJECT _ID = ”`patriotic-F39706271-1764804660`”

Rules:

- PROJECT _ID MUST be generated for each stack.

- A new PROJECT _ID MUST NOT be reused across different deploys.
- PROJECT _ID is written into tags/labels so that any running resource can be tied back to a specific release.
- Used for billing, reporting, and governance — **never** overloaded to mean a cloud provider project or a container deployment.

Once chosen, PROJECT_ID MUST remain stable for the life of the engagement.

11.1.3) STACK (Change Management (CM) Environment Identity)

STACK identifies a specific **environment instance** of an engagement (for example, the WordPress dev stack for a given project).

Logical format: STACK = "<APP_SLUG>-<CM_ENV>"

Where:

- APP_SLUG – short, lowercase, human friendly slug for the application or template (e.g. bonsai , driver)
- CM_ENV (e.g. dev, qas, prd)

11.1.4) DEPLOYMENT_ID (Deployment Identity)

Uniquely identifies a **specific deployment instance** of a project and is set by default to the PROJECT_ID. The DEPLOYMENT_ID should match the target environment's project ID if that value is in our control. If not, the DEPLOYMENT_ID should reflect the target environment's project ID,

Format & Constraints:

- 6–30 characters
- [a-z0-9-] only
- Must start with [a-z]
- Must end with [0-9]

Example:

- PROJECT_ID = "patriotic-F39706271-1764804660"
- DEPLOYMENT_ID = "patriotic-F39706271-1764804660"

Rules:

- DEPLOYMENT_ID defaults to PROJECT_ID though may be overwritten to represent target project id
- A new DEPLOYMENT_ID MUST NOT be reused across different deploys.
- DEPLOYMENT_ID is written into tags/labels so that any running resource can be tied back to a specific release.

Example : tesla-inspired-info-dev

11.1.5) Environment Files

Environment files are **operator-side artifacts**, never loaded directly by cloud runtimes.

11.1.6) Naming Convention

- Development:
  - `DEV` → `.env`
- Non-development environments:
  - `QAS` (Quality Assurance) → `qas.env`
  - `PRD` (Production) → `prd.env`
  - `PSE` (Production Support Environment) → `pse.env`
  - Additional environments (e.g., `SIT`, `UAT`) MAY be added with same pattern (e.g., `sit.env`, `uat.env`).

11.1.7) Scope and Usage

- .env:
  - Used for **local development**.
  - Consumed by local scripts (init-local-wp.sh)
  - MUST NOT be used by production runtimes.
- *.env: (e.g. qas.env, prd.env, pse.env):
  - Used by operators and scripts to configure remote environments.
  - Drive pm-provision-gcp, pm-setup, etc.
  - Production runtimes receive config via platform env and secrets, not via –-env-file

11.1.8) Required Keys (by convention)

Common keys across env files include:

- identity:
  - APP_SLUG
  - CM_ENV
  - PROJECT_ID (canonical engagement ID, required)
  - STACK (derived or explicitly set)
  - DEPLOYMENT_ID (derived or explicitly set)
- Routing:
  - LOCAL_DOMAIN (e.g. localtest.me)
  - APP_HOST (derived or explicitly set)
- Database:
  - DB_NAME
  - DB_USER
  - DB_PASSWORD

    - DB_ROOT_PASSWORD
    - CLOUDSQL_INSTANCE
- WordPress:
    - URI_SCHEME
    - WP_SITE_URL
    - WP_ADMIN_USER
    - WP_ADMIN_PASS
    - WP_ADMIN_EMAIL
    - WP_SITE_TITLE
    - WP_THEME_SLUG
- Tooling:
    - RUNTIME_ADAPTER_CODE
    - TRAEFIK_DIR
    - HTTP_HOST

11.2) Local Development

Local environments use:

- Traefik as a shared ingress proxy on a Docker proxy network
- localtest.me (or equivalent) as the development domain.
- docker-compose.yml plus an environment file (.env)

`init-local-wp.sh` responsibilities:

11.2.1) Ensure `.env` exists (scaffold from `.env.example` if present).
11.3.2) Load `.env`.
11.3.3) Derive and persist `STACK` (from `APP_SLUG`, `PROJECT_ID`, `CM_ENV`) and, where appropriate, `CLOUD_PROJECT_ID` where missing.
11.3.4) Copy `docker-compose-local-wp.yml` to `docker-compose.yml` if no override exists.
11.3.5) Call `provision-site.sh` to register the site with Traefik and handle TLS where required.
11.3.6) Bring up the local WordPress + MariaDB containers.
11.3.7) Run `configure-site.sh` and `publish-content.sh` (NOTE: `configure-site.sh` should call `configure-site-boilerplate.sh` first. `publish-content.sh` shares the same pattern.)

`init-local-wp.sh` MUST be idempotent; re-running it SHOULD leave a running site unchanged except where deliberate updates are intended.

11.3) Cloud Environments (GCP)

GCP environments (QAS, PRD, PSE) are bootstrapped via:

11.4.1) `scripts/init-gcp-wp.sh --env qas.env|prd.env|pse.env`
11.4.2) `scripts/deploy-gcp-wp.sh --env qas.env|prd.env|pse.env`
11.4.3) `scripts/configure-gcp-site.sh --env qas.env|prd.env|pse.env`

11.3.1) init-gcp-wp.sh

- Validates that the specified .env file exists and can be sourced.
- Ensures `STACK` (logical) is present or derives it from `APP_SLUG`, `PROJECT_ID`, and `CM_ENV`.
- Ensures or derives `CLOUD_PROJECT_ID` and persists it back into the env file.
- Creates the GCP project (`CLOUD_PROJECT_ID`) if it does not yet exist.
- Optionally links billing (when `GCP_BILLING_ACCOUNT` is set).
- Enables required services:
  - `run.googleapis.com`
  - `sqladmin.googleapis.com`
  - `artifactregistry.googleapis.com`
  - `cloudbuild.googleapis.com`
  - and any others required by the project.

11.3.2) deploy-gcp-wp.sh

- Reads the env file for:
  - `CLOUD_PROJECT_ID`, `STACK`, `REGION`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`.
- Computes `stack_slug` from `STACK` where required for constrained resource names.
- Creates or ensures an Artifact Registry repo for container images.
- Builds a WordPress image from `Dockerfile-gcp-wp` and pushes it.
- Creates or ensures a Cloud SQL instance for MySQL (e.g., `{stack_slug}-sql`).
- Creates/ensures the database and user.
- Deploys a Cloud Run service named `{stack_slug}-wp`, configured to:
  - Use the built image.
  - Attach to the Cloud SQL instance using `/cloudsql/{CLOUD_PROJECT_ID}:{REGION}:{stack_slug}-sql`.
  - Set environment variables from the env file (DB config, STACK, PROJECT_ID, etc.).
- Retrieves and persists the Cloud Run URL back into the env file as `CLOUDRUN_URL`.

11.3.3) configure-gcp-site.sh

- Uses the .env file and `CLOUDRUN_URL` to configure the remote WordPress site.
- Establishes a connection to Cloud SQL via `cloud-sql-proxy`.
- Runs `configure-site-boilerplate.sh` and `configure-site.sh` locally using WP-CLI, but targeting:
  - the remote Cloud SQL database, and
  - the Cloud Run public URL as the site URL.

This ensures that non-exported properties (language, permalink structure, menus, etc.) are configured consistently in each environment.

11.4) Production Rules

11.4.1) No .env files in runtime

- Production runtimes (GCP, AWS, Azure, etc.) MUST NOT use `--env-file` or mount `.env` files.
- Configuration MUST be provided via:
  - Platform environment variables, and
  - Secret managers (Cloud Secret Manager, AWS Secrets Manager, etc.).

11.4.2) Identity Invariance

- Once an environment's `PROJECT_ID`, `STACK`, and `CLOUD_PROJECT_ID` are set and persisted in its env file, they MUST NOT be overridden transiently.
- Commands like `CLOUD_PROJECT_ID=foo gcloud run deploy ...` or `STACK=bar docker compose up ...` are prohibited.
- If identity must change, the corresponding .env file MUST be edited and the change applied consistently across all tooling.
- `DEPLOYMENT_ID` MUST be treated as **append-only** history: a new `DEPLOYMENT_ID` is generated per deploy; existing `DEPLOYMENT_ID` values are never reused or retroactively changed on running resources.

11.4.3) Local Docker Production

In limited cases where production or near-production is hosted on a local Docker host under direct operator control:

- An env file (`prd.env` or similar) MAY be used to drive `docker-compose`.
- That file MUST NOT be checked into version control.
- The same identity invariance and secrets discipline rules apply:
  - No ad-hoc overrides at runtime.
  - Changes are made by editing the env file and re-applying.

11.5) Relationship to EPC

This document implements EPC's requirements in the environment and deployment domain:

- Identity via `PROJECT_ID` (engagement), `STACK` (environment), `DEPLOYMENT_ID` (release), and `CLOUD_PROJECT_ID` (cloud provider project).
- Namespacing of hosts and services.
- 12-factor configuration.
- Platform-managed secrets in production.
- No manual edits between environments; scripts and templates consume env and generate artifacts.
- A single ingress/front-door per stack.

For the full normative statement of EPC, see **Appendix A**.

12) Artifact Verification & Integrity

This section defines how artifacts (zips, containers, binaries, documents) are created, identified, and verified.

12.1) Goals

- Ensure that anyone consuming an artifact can:
  - Verify its integrity.
  - Confirm which version and source it corresponds to.
  - Reproduce it if given the source repo and commit.

12.2) Hashes and Metadata

12.2.1) SHA-256 Requirement

- Every downloadable artifact produced for customers or internal reuse MUST publish:
  - File name.
  - File size in bytes.
  - SHA-256 hash.
- These MAY be published:
  - In a `CHECKSUMS.txt` file alongside the artifacts.
  - In release notes.
  - In documentation accompanying the artifact.

12.2.2) Example

```
TGF.zip
  size: 123456 bytes
  sha256: 3f2c1a0b... (64 hex chars)
```

Scripts that generate artifacts SHOULD compute and print these values to simplify compliance.

12.3) Container Images

- Container images MUST be built from a known source state (git commit or tag).
- Image tags SHOULD include:
  - A human friendly label (e.g., `v1.5.0`), and
  - A unique immutable identifier (e.g., short git SHA or timestamp).
- Where platforms support it, images SHOULD also be signed (e.g., with Cosign) and signatures stored in a public or internal registry.

12.4) Reproducibility

- Build pipelines SHOULD be defined as code and checked in with the project.
- Dependencies SHOULD be pinned (lockfiles, exact versions) so that builds are reproducible.
- Any non-determinism (timestamps, random IDs) SHOULD be isolated or documented.

12.5) Verification in CI/CD

- CI/CD pipelines SHOULD include a verification step that:
  - Confirms artifacts were built from the expected commit.
  - Computes and records hashes.
- For high-risk deployments, a manual verification step (comparing expected hash vs. actual) MAY be required by operations before release.

12.6) Relationship to EPC

EPC requires that deployments be:

- Idempotent.
- Reproducible.
- Verifiable.

This section describes how artifacts contribute to those properties. Artifacts that cannot be tied back to a known source and verified are considered non-compliant.

Appendix A) Et al Portable Contract (EPC)

EPC defines how Et al Solutions LLC designs systems so that they remain portable across runtimes (local Docker, Kubernetes, Cloud Run, ECS, WordPress.com where applicable) without changing application code.

A1) Statement of Principle

With EPC guidelines and guardrails, teams only need to implement a **runtime adapter** for any supported platform. Application code and configuration patterns remain consistent; the adapter handles wiring, secrets, ingress, and platform specifics.

EPC aims to ensure that:

- Identity is unambiguous and collision-free (`PROJECT_ID`, `STACK`, `DEPLOYMENT_ID`).
- Configuration follows 12-factor principles.
- Secrets are managed by the platform, not embedded in code or env files.
- Deployments are idempotent and can be safely re-run.
- Porting a workload between runtimes requires only adapter changes, not code changes.

A2) Core Tenets

A2.1) Identity & Namespacing

- All externally visible names (hosts, namespaces, services, buckets, queues, topics) MUST include a unique STACK identifier (e.g., `wp-F39797271-dev`, `bonsai-dev`, `bonsai-qas`).
- Resources MUST NOT share names across environments or customers.

A2.2) 12-Factor Configuration

- Applications read configuration exclusively from environment variables.
- No environment-specific code branches; differences are expressed via env values.
- `.env` files are allowed only on the operator side, never as production runtime dependencies.

A2.3) Secrets Discipline

- Secrets are never committed to version control.
- Production secrets are injected from the platform (Secret Manager, Key Vault, Parameter Store, etc.).
- Development MAY use `.env` for convenience, but production MUST NOT.

A2.4) Single Front Door

- Each stack exposes HTTP(S) entry through a single ingress:
  - Traefik for local development.
  - Cloud Run / Load Balancer / API Gateway / Front Door in cloud environments.
- Backends do not bind public host ports directly.

A2.5) Safety Rails

- Production deployments MUST require an explicit ARM toggle or manual approval gate.
- Deployment scripts MUST fail fast when preconditions are not met.
- Scripts MUST be safe to re-run (idempotent) unless explicitly documented.

A2.6) Observability by Convention

- Health endpoints and log/metric labels MUST include STACK.
- Where appropriate, PROJECT_ID or DEPLOYMENT_ID SHOULD also be included.

A2.7) Zero Manual Edits Across Environments

- No hand-editing files between customers or environments.
- Scripts and templates consume env files and render artifacts automatically.
- Any environment-specific overrides are captured in env files (`.env` , `qas.env` , `prd.env` , etc.).

A2.8) EPC Tag Schema (new in v1.7)

All EPC runtimes MUST apply a standard tag/label schema using the namespace: solutions.etal.*

This replaces any previous `com.etalsolutions.*` namespace. Existing tags using the old namespace MUST be migrated to the `solutions.etal.*` keys over time.

A2.8.1) Identity Tags

Key	Description	Required
`solutions.etal.project_id`	Canonical Et al engagement ID (`PROJECT_ID`)	YES
`solutions.etal.stack`	Logical stack identity (`STACK`)	YES
`solutions.etal.deployment_id`	Unique deployment instance (`DEPLOYMENT_ID`)	YES

A2.8.2) Service Classification Tags

Key	Description	Allowed Values
`solutions.etal.service`	Service role/classification	`api`, `web`, `worker`, `db`, `cache`, `queue`, `proxy`, `cron`, `client-build`, `ephemeral`, `other`
`solutions.etal.service_tier`	Tier of responsibility	`frontend`, `backend`, `persistence`, `edge`, `internal`, `ephemeral`
`solutions.etal.managed_by`	Primary owner/ops responsibility	`etal`, `customer`, `third-party`
`solutions.etal.service_lifecycle`	Lifecycle state	`active`, `deprecated`, `retired`, `experimental`

A2.8.3) Data Governance Tags

Key	Description	Allowed Values
`solutions.etal.data_class`	Data sensitivity	`public`, `internal`, `confidential`, `restricted`
`solutions.etal.data_lifecycle`	Retention / lifecycle	`ephemeral`, `short`, `medium`, `long`, `archival`

A2.8.4) Application

- Docker Compose services MUST apply these as labels where supported.
- Kubernetes workloads MUST carry these as labels/annotations on Pods, Services, and Ingress where appropriate.
- Cloud Run services and jobs MUST be labeled with this schema where platform capabilities permit.
- Messaging and data services (e.g., Pub/Sub topics, queues, buckets) SHOULD be annotated or documented using this schema via IaC or platform tags.

A3) Non-Negotiables (SOP Snippets)

- No hard-coded hostnames. Hosts are `${STACK}.${LOCAL_DOMAIN}` (dev) and `${STACK}.${BASE_DOMAIN}` (prod).
- No `--env-file` in production anywhere (scripts, CI, manifests).
- ARM toggle required (e.g., `ops/ALLOW_PROD_DEPLOY.on`) before any prod deploy.
- Idempotent infra: rerunning init/build/deploy must be safe.
- Everything namespaced by STACK.

A4) Adapter Pattern (How We Apply EPC)

For each runtime (Docker Compose, Kubernetes, Cloud Run/App Runner/ACA/DO, Functions, etc.), we provide a small adapter that:

- wires the edge (host → service) using `${STACK}` hosts,
- maps standard env names (e.g., `WORDPRESS_DB_HOST`, `WORDPRESS_DB_USER`, …),
- references platform secrets for prod,
- preserves EPC non-negotiables.

A5) Boilerplate (verbatim for SOW/README)

> **Et al Portable Contract (EPC).** This project adheres to EPC: identity via STACK namespacing; 12-factor configuration; platform-managed secrets in production; a single ingress with host-based routing; explicit ARM gating for production deploys; and idempotent, zero-edit automation across environments. To target a new runtime, we supply a minimal adapter without changing application code.

A6) How We Keep It Zero-Touch & Repeatable

- **No hard-coded hosts anywhere.** Hosts are computed:
  - **Dev (Compose):** Use a project-root `.env` with **concrete** values to avoid Compose interpolation pitfalls. Example → `STACK=acctdemo`, `LOCAL_DOMAIN=localtest.me`, `APP_HOST=acctdemo.localtest.me`.
  - **Prod:** `APP_HOST=${STACK}.${BASE_DOMAIN}` is computed via CI/platform environment (never stored in a file).
- **Dev env files:**
  - **Root `.env` (Compose interpolation source):** concrete values for `STACK`, `LOCAL_DOMAIN`, `APP_HOST`, and any **non-secret** dev DB values.
  - Optional `env/${STACK}.env` for scripts/tooling; keep it consistent with `.env`.
- **Scripts load env & apply templates** (envsubst, Helm/Kustomize vars, Terraform variables). In dev, prefer root `.env` for Compose interpolation; use `--env-file` only for container-level env where appropriate.
- **ARM toggle enforced** in any prod script regardless of platform.
- **Standard env names** across all runtimes (`WORDPRESS_DB_*`, `WP_*`, `APP_HOST`, `STACK`) so app code never changes.

A7) Base Templates

- New WordPress projects SHOULD start from the **wp-with-mariadb** base template:
  - Includes `init-local-wp.sh`, `configure-site-boilerplate.sh`, `configure-site.sh`, `publish-content-boilerplate.sh`, `publish-content.sh`, and a standard `docker-compose-local-wp.yml`.
  - Uses EPC-compliant environment variable names.

A8) sys-clone-project

- Projects SHOULD be cloned using a `sys-clone-project` utility that:
  - Copies the directory tree.
  - Excludes any previous `.project_timestamp`.
  - Creates a new `.project_timestamp` in the target folder.

A9) init Scripts

- Each environment (local, GCP, etc.) MUST provide an `init-*.sh` script that:
  - Ensures the appropriate environment configuration exists (`.env`, `qas.env`, etc.).
  - Loads, validates, and writes back to configuration, STACK and other identities (deriving them if missing).
  - Boots the local or remote runtime (Docker, Cloud Run, etc.).
  - Is idempotent whenever possible.

A10) Documentation

- Each repo MUST include a `README.md` describing:
  - Purpose of the project.
  - How to bootstrap local development.
  - How to run tests.
  - How to deploy (or a link to the appropriate SOP).
- For repeatable operations (backups, migrations, cutovers), a `RUNBOOK.md` SHOULD be created.

A11) Exceptions

- Exceptions to these standards MUST be documented in the repo and, where relevant, approved in writing.
- Customer-mandated deviations (for example, forced branch naming) SHOULD be isolated to that customer's repos and not propagated as defaults.

A12) Runtime Adapters

A **runtime adapter** is a small, environment-specific layer that maps the EPC contract to a particular runtime:

- Docker Compose.
- Kubernetes.
- GCP Cloud Run + Cloud SQL.
- AWS ECS / App Runner / Lambda.
- Azure Container Apps / Functions.
- WordPress.com (where applicable, via APIs and export/import).

Adapters MUST:

- Use STACK and other env values for namespacing.
- Configure ingress correctly.
- Bind services to platform secrets and configuration mechanisms.
- Avoid introducing new identity or configuration sources.

Current Runtime Adapters (Same Contract, Different Runtime)

A12.1) Kubernetes (+ managed DB)

Collision-proofing:

- Namespace per stack: `kubectl create ns ${STACK}`
- Ingress host: `${STACK}.${LOCAL_DOMAIN}` (dev) / `${STACK}.${BASE_DOMAIN}` (prod)

Example (templated with envsubst):

```
# k8s/deploy.yaml.tpl
apiVersion: apps/v1
kind: Deployment
metadata: { name: app, namespace: ${STACK}, labels: { app: ${STACK}-app } }
spec:
  replicas: 1
  selector: { matchLabels: { app: ${STACK}-app } }
  template:
    metadata: { labels: { app: ${STACK}-app } }
    spec:
      containers:
      - name: app
        image: ${IMAGE}
        env:
        - { name: WORDPRESS_DB_HOST, value: "${DB_HOST}" }
        - { name: WORDPRESS_DB_USER, value: "wp_user" }
        - { name: WORDPRESS_DB_NAME, value: "wordpress" }
        - { name: WORDPRESS_DB_PASSWORD, valueFrom: { secretKeyRef: { name: ${STACK}-secrets, key: DB_PASS } } }
        ports: [{ containerPort: 8080 }]
---
apiVersion: v1
kind: Service
metadata: { name: app-svc, namespace: ${STACK} }
spec:
  selector: { app: ${STACK}-app }
  ports: [{ port: 80, targetPort: 8080 }]
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata: { name: app-ing, namespace: ${STACK}, annotations: { kubernetes.io/ingress.class: traefik } }
spec:
  rules:
  - host: ${APP_HOST}
    http:
      paths:
      - path: /
        pathType: Prefix
        backend: { service: { name: app-svc, port: { number: 80 } } }
```

Render & apply (no file edits):

```
export STACK=summitdemo APP_HOST=${STACK}.local.test IMAGE=ghcr.io/org/app:dev DB_HOST=mysql.default.svc:3306
envsubst < k8s/deploy.yaml.tpl | kubectl apply -f -
```

Secrets:

```
kubectl -n ${STACK} create secret generic ${STACK}-secrets --from-literal=DB_PASS=...
```

A12.2) Serverless Containers (Cloud Run / App Runner / ACA / DO App Platform)

Edge: platform domain mapping; host = `${STACK}.${BASE_DOMAIN}`
DB: managed service (Cloud SQL / RDS / Azure MySQL / DO Managed MySQL)
Env/secrets: set via platform CLI; **never** `--env-file` in prod

Deploy skeleton (cloud-agnostic):

```
STACK=${STACK:?} IMAGE=${IMAGE:?} APP_HOST=${STACK}.${BASE_DOMAIN}
DB_HOST=${DB_HOST:?} SECRET_REF=${SECRET_REF:?}

# GCP example
gcloud run deploy ${STACK}-app --image "$IMAGE" --region "$REGION" --project "$CLOUD_PROJECT_ID" --quiet
gcloud run services update ${STACK}-app --region "$REGION" --project "$CLOUD_PROJECT_ID"     --set-env-vars WORDPRESS_DB_HOST="$DB_HOST",
```

A12.3) Static Front-End + Serverless API

- Edge: same host derivation (`${STACK}.${BASE_DOMAIN}`), subpaths/subdomains for API.
- Dev: one local proxy routes `/api` → local function/container; no ports exposed.
- Collision-proofing: route prefixes include `${STACK}` or use per-stack hosts.

A12.4) Functions (Lambda / Cloud Functions / Azure Functions)

- Hostnames: behind API Gateway / Front Door—use per-stack stage or subdomain.
- Env/secrets: always platform; local dev uses `--env-file` via emulator only.
- Collision-proofing: stage names or routes include `${STACK}` .

A12.5) Data/Messaging Pipelines

- Resource names: topics/queues/streams include `${STACK}` (e.g., `${STACK}.events`).
- Secrets/config: env-injected; dev uses `.env` file, prod from platform.
- No ports/hosts exposed—still keep the naming & ARM rules.

A12.6) Docker Compose (Local Dev) — New in v1.1

- Edge: Traefik on shared `proxy` network; router rule `Host(${APP_HOST})` .
- Identity: Everything namespaced by `STACK` ; containers/volumes scoped by project.
- Config: Root `.env` with **concrete** values: `STACK=acctdemo` , `LOCAL_DOMAIN=localtest.me` , `APP_HOST=acctdemo.localtest.me` . Avoid nested `${…}` inside `.env` .
- Secrets: Dev may use `.env` for non-secrets and local secrets; **never** in prod.
- Pattern: `docker-compose.yml` defines `wordpress` , `db` , and `wpcli` services sharing volumes; MU plugins provide AI/SEO readiness and one-time child theme activation; `init-wp.sh` sources env, installs WP if needed, activates child theme, sets permalinks.
- Collision-proofing: `APP_HOST` includes `STACK` (e.g., `acctdemo.localtest.me`). Backends do not bind host ports; all traffic via Traefik.

A13) Runtime Adapter Checklist

1. Edge: define how `${APP_HOST}` is bound (Ingress, LB, Gateway).
2. Runtime: container service / functions / k8s; confirm target port and health.
3. Config: map the standard env names to platform deployment knobs.
4. Secrets: map `WORDPRESS_DB_PASSWORD` to platform secret reference.
5. Persistence: set `${DB_HOST}` (socket path on GCP; TCP host:port elsewhere).
6. Namespacing: ensure every resource name includes `${STACK}` .
7. Dev parity: local proxy + `--env-file` ok in dev; prod forbids `--env-file` .
8. Safety: ARM toggle check at the top of prod scripts.
9. Compose specifics: ensure a **root** `.env` exists with **concrete values** for interpolation.

A14) Dev vs Prod Clarifications (v1.1)

- Dev (Compose): root `.env` ignored by git; optional `env/${STACK}.env` for scripts.
- Prod: no `.env` files; no `--env-file`; env & secrets injected by platform/CI.

A15) WordPress Example

For WordPress projects, EPC manifests as:

- A base template (`wp-with-mariadb`) with:
  - `sys-clone-project` for stamping new stacks (`.project_timestamp`).
  - `init-local-wp.sh` for local Docker + Traefik.
  - , `deploy-gcp-wp.sh` , `configure-gcp-site.sh` for GCP.

    `init-gcp-wp.sh`
- Environment files:
  - `.env` for DEV.
  - `qas.env` , `prd.env` , `pse.env` for higher environments.
- Shared configuration scripts:
  - `configure-site-boilerplate.sh` and `configure-site.sh` .
  - `publish-content-boilerplate.sh` and `publish-content.sh` .
- Export/import as the content-movement mechanism, layered on top of EPC-compliant runtime configuration.

By adhering to EPC, the same base project can be hosted locally, on GCP (Cloud Run + Cloud SQL), and, with appropriate adapters, on other platforms without changing application code.

A16) Compliance

Projects are considered **EPC-compliant** when:

1. Identity and namespacing rules are followed.
2. Configuration and secrets rules are followed.
3. Runtime adapters are used rather than inlining platform logic throughout the codebase.
4. Deployment flows are idempotent, observable, and verifiable.

Non-compliant projects SHOULD be brought into alignment as they are touched for maintenance or new feature work.

Appendix B) Unified Naming Conventions

Scope: Software (C#/.NET, repos, APIs, DB), Azure resources, CI/CD, infra

B1) Azure Resources

General rule: `<resource-type-abbr>-<env>-<app/feature>-<instance>`
Env codes: `p` = prod, `d` = dev, `t` = test, `s` = staging

Examples:

- `rg-p01-payments` (Resource group)
- `stp-d01-data` (Storage account)
- `app-s01-identity` (App service)

Use lowercase, numbers, and hyphens. No special chars.
Keep names globally unique where required (storage, DNS).

B2) C# / .NET Code

B2.1) General

- PascalCase for: namespaces, classes, structs, records, enums, delegates, public members.
- camelCase for: local variables, method parameters.
- Private fields: `_camelCase`.
- Static fields: `s_camelCase`.
- Thread-static: `t_camelCase`.
- Constants: PascalCase.
- Interfaces: Prefix with `I` (e.g., `IWorkerQueue`).
- Attributes: Suffix with `Attribute`.
- Enums: Singular noun (plural if `[Flags]`). Members in PascalCase.
- Booleans: Prefix with `Is/Has/Can/Should`.
- Async methods: Suffix with `Async`.
- Events: Noun/verb phrase (`OrderProcessed`, `Processing`).
- Acronyms: `Id`, `Http`, `Xml` (not all caps).
- Files: One public type per file; filename = type name.

B2.2) Type Parameters

- Prefix with `T` (e.g., `TSession`).
- Use descriptive names unless a single letter is obvious (`T`, `K`, `V`).

B2.3) Cross-Boundary Naming

- JSON (APIs/config): camelCase.
- HTTP paths: kebab-case, lowercase. Example: `/customer-orders/{orderId}`.
- Environment variables: SCREAMING_SNAKE_CASE.
- Database: snake_case for tables/columns. Map cleanly from C# PascalCase → DB snake_case.

B2.4) Repos & Packages

- Solution/Assembly/Namespace: `Company.Product.Component`.
- NuGet packages: `Company.Product.Component`.
- Docker images: `company/product-component:tag` (lowercase, dash-separated).
- Git branches: kebab-case with prefix (`feat/add-customer-search`).
- Commits: Conventional Commits (`feat:`, `fix:`, `chore:`).

B2.5) CI/CD & Infra

- YAML jobs/steps: kebab-case.
- Terraform/ARM/Bicep modules: snake_case.
- Secrets/keys: SCREAMING_SNAKE_CASE.

B2.6) Enforcement

- Use `.editorconfig` with Roslyn analyzers for casing, prefixes, and async suffix.
- Configure `System.Text.Json` to enforce camelCase JSON.
- Lint PRs for non-timestamped logs and naming drift.

B2.7) Quick Examples

```
public interface IWorkerQueue
{
}

public class DataService
{
    private readonly IWorkerQueue _workerQueue;
    private static ILogger s_logger;
    public bool IsEnabled { get; set; }

    public async Task ProcessAsync(int orderId)
    {
        var retryCount = 0;
        await _workerQueue.EnqueueAsync(orderId);
    }
}
```

JSON API:

```
{ "orderId": 123, "customerName": "Alice" }
```

Azure resource:

```
rg-p01-payments
```

Branch name:

```
feat/add-customer-search
```

B2.8) Exceptions

- Third-party generated code may differ; don’t manually alter.
- Structured JSON logs may omit human prefixes if timestamp field is present.

Appendix C) EPC Service Classification & Runtime Model

This section defines how we classify all software services and clients within Et al Solutions LLC under the Et al Portable Contract (EPC). The goal is to keep a **small, finite set of patterns** that cover all current and future technologies without creating a combinatorial explosion.

All systems are described along **three axes**:

1. **Role/Responsibility** – What the component does.
2. **Runtime** – What execution environment it runs on.
3. **Adapter** – How and where it is hosted or built.

Together, these three layers fully describe any service or client we build.

C1) Roles

Roles describe the **role/responsibility** of a component.

C1.1) Server-side role/responsibility (EPC runtimes)

C1.1.1) API

HTTP JSON services. Examples: booking engines, backends-for-frontend, authentication APIs.

C1.1.2) Web

Browser-facing websites and applications. Includes static sites, SPAs, and SSR/fullstack web apps.

C1.1.3) Worker

Background processes and jobs. Includes queue consumers, scheduled tasks, data pipelines, and batch processors.

C1.2) Client-side role/responsibility (EPC-aware, not an EPC runtime)

C1.2.1) Client App

Mobile or desktop applications that consume EPC APIs. Examples: iOS, Android, cross-platform mobile (Flutter, React Native), and desktop clients. Client Apps are **built and distributed**, not hosted as EPC runtimes, but must be EPC-aware when calling backend services.

C2) Runtimes

Runtimes describe the **execution environment**, not the framework. Frameworks (React, Next.js, Laravel, Django, etc.) always sit inside one of these runtime families.

C2.1) Server runtimes (finite set)

C2.1.1) node

JavaScript/TypeScript runtime.
Examples: Express, Nest, Next.js, Remix, SvelteKit, Vite SSR.

C2.1.2) php

PHP runtime.
Examples: Laravel, Symfony, WordPress, Slim.

C2.1.3) python

Python runtime.
Examples: Django, Flask, FastAPI, Celery workers.

C2.1.4) java

JVM runtime.
Examples: Spring Boot, Micronaut, Quarkus.

C2.1.5) dotnet

.NET runtime.
Examples: ASP.NET Core APIs, Blazor Server, .NET worker services.

C2.1.6) static

Built frontends (HTML/CSS/JS) served from object storage and/or CDN.
Examples: React/Vue/Angular/Svelte builds, Astro static output.

C2.2) Client runtimes (build targets, not EPC runtimes)

For the **Client App** role we recognize:

C2.2.1) ios

Swift/SwiftUI/UIKit-based apps.

C2.2.2) android

Kotlin/Java-based apps.

C2.2.3) xplat-mobile

Cross-platform mobile (Flutter, React Native, etc.).

Client runtimes are **build targets**. They are not EPC runtimes in our infra, but they **must be EPC-aware** when configuring API endpoints.

C3) Adapters

Adapters define **where and how** a runtime is hosted or built.

C3.1) Server-side adapters

All server-side adapters must obey EPC:

- All externally visible names (hosts, namespaces, services) include `STACK`.
- Configuration comes from environment variables (12-factor).
- Production secrets are provided by the platform (no `--env-file` in prod).
- There is a single front door per stack (host-based routing).
- Infra and deploy scripts are idempotent and ARM-gated for prod.

C3.1.1) dev-compose

Local development using `docker compose` and a shared `proxy` network.

- Services are accessible via `${STACK}.${LOCAL_DOMAIN}`.
- Traefik (or equivalent) provides a single front door in dev.

C3.1.2) cloudrun

Google Cloud Run (containerized HTTP services or jobs).

- Public hosts use `${STACK}.${BASE_DOMAIN}`.
- Env configuration and secrets come from the platform.

C3.1.3) k8s (future)

Kubernetes-based deployments.

- Ingress terminates traffic for `${STACK}.${BASE_DOMAIN}` and routes to namespaced services.

C3.1.4) functions/serverless (optional)

Function-as-a-service or job-style runtimes (Cloud Functions, Cloud Run Jobs, etc.), still obeying EPC identity and config rules.

C3.2) Client-side adapters (distribution)

Client Apps are distributed, not hosted:

C3.2.1) appstore

Apple App Store / TestFlight.

C3.2.2) playstore

Google Play / internal tracks.

C3.2.3) enterprise-distribution

MDM or internal app distribution.

C3.2.4) web-build

Build pipelines for PWAs or browser-based clients.

Client Apps must:

- Use an API base URL derived from `STACK` and the current environment, for example:
  - Dev: `https://${STACK}.${LOCAL_DOMAIN}`
  - Prod: `https://${STACK}.${BASE_DOMAIN}`
- Never hard-code production URLs directly into source; configuration must be environment-driven.

C4) Putting It Together

This finite classification avoids a "runtime/framework zoo" and ensures that any new technology choice (e.g., SvelteKit, Remix, Blazor, Next.js) fits cleanly into the existing EPC framework without expanding the runtime surface area.

C4.1) Every server-side component is fully described by:

Role + Runtime + Adapter

Examples:

- `API + node + cloudrun` – Ascend Booking API written in Node.js and deployed to Cloud Run.
- `Web + static + cloudrun` – Static SPA built in React, served from a container or GCS+CDN.
- `Worker + dotnet + cloudrun` – .NET background worker deployed as a Cloud Run service or job.

C4.2) Every client-side app is described by:

Role = Client App + Client Runtime + Distribution Adapter

Examples:

- `Client App + ios + appstore` – iOS app consuming EPC APIs.
- `Client App + xplat-mobile + appstore/playstore` – Flutter or React Native app consuming EPC APIs.

Appendix D) CHANGELOG

Technical Governance Framework (TGF) v1.8.1 — Changelog

Date: 2025-12-17

v1.8.1 (Current)

- Added
  - Standardized shell control-flow + logging primitives (`bail` , `fatal` , `ok` , `warn` , `die` , `fatal_error`) to distinguish “exit this layer” vs. “red button” behavior across sys-, *ops*-, and adapter tooling.

v1.8

- Added
  - Safe-return semantics for shell functions in Section 8.2 (Shell). Functions used in sys-, *ops*-, and adapter tooling now return values via caller-provided variables (using `printf -v`) instead of stdout.
  - Explicit stdout/stderr discipline language for shell functions whose output is consumed by other scripts or pipelines.
- Updated
  - Section 8.2 (Shell) to clarify that shell scripts MUST treat stdout as machine data and stderr as the exclusive channel for human-readable logs.
- No Changes
  - EPC non-negotiables and runtime adapter expectations remain unchanged.

v1.7 (Prior)

- Added
  - Node.js Runtime Adapter Reference
    Introduced a full adapter reference for Node.js projects, aligned with the existing WordPress adapter reference.
  - Documents EPC-compliant host derivation, env mapping, and secrets discipline for Node.js apps.
    Defines standard mappings for APP_HOST, STACK, and common server environment variables.
  - Provides dev/prod parity rules, including local proxy expectations, `--env-file` usage constraints, and platform-managed secrets for production.
    Establishes baseline scaffolding guidance for Express/Fastify-style services while remaining runtime-agnostic.
  - EPC Tag Schema (`solutions_etal_*`) under Appendix A2.8, adding explicit identity, service classification, and data governance tags.
  - Clarified identity model across EPC and environment standards: `PROJECT_ID` (engagement), `STACK` (environment), `DEPLOYMENT_ID` (deployment), `CLOUD_PROJECT_ID` (provider project).
- Updated
  - Adapter Reference Index
  - Identity and namespacing language in EPC sections and Environment Standards.
- No Changes
  - No revisions to EPC non-negotiables or base WordPress adapter behavior.

v1.6 (Prior)

- Added **environment file naming convention**:
  - `DEV = .env`
  - `QAS = qas.env`
  - `PRD = prd.env`
  - `PSE = pse.env` (Production Support Environment)
- Codified rule that:
  - `*.env` files are **operator-side inputs only**.
  - Cloud production runtimes MUST receive configuration via platform-managed environment variables and secrets.
- Clarified **STACK** and **PROJECT_ID** rules and how they are derived and persisted.
- Described the **WordPress + MariaDB reference flow**:
  - `sys-clone-project` → `.project_timestamp`
  - `init-local-wp.sh` → `.env` + identity + local Traefik + WP
  - GCP adapters for Cloud Run + Cloud SQL

v1.5 (Prior)

- Tightened **Artifact Verification & Integrity**:
  - All downloadable artifacts MUST publish size and SHA-256.
- Updated EPC reference to reflect:
  - Identity via `STACK` namespacing.
  - Ban on `--env-file` in production.
  - Standard runtime adapter expectations (Docker, Cloud Run, etc.).

v1.4 (Prior)

- Introduced EPC as a first-class governance document.
- Defined STACK-based namespacing for hosts, queues, buckets and services.
- Standardized testing matrix for JS/TS and PHP projects.
- Established Traefik + `localtest.me` as the default local HTTP entrypoint.

v1.3 and earlier

- Initial consolidation of scattered standards into the TGF series.
- Early forms of development, testing, and CI standards.
- Pre-EPC environment and deployment conventions.