ballastlane
APPLICATIONS

CAPABILITY BRIEF

BLA's Marshal: Framework for Advanced AI-enabled Teams

Adding governed, cross-provider rigor to an established AI development practice, without replacing what works.

Who this is for

You already have an AI-enabled development practice. Your engineers use AI coding tools, you've tuned a setup that works for you, and you run a brownfield production system with real users and no tolerance for breakage. You don't need a vendor to stand up an engineering organization — you have one.

This document is about what Ballast Lane Applications (BLA) adds to a practice like yours, and where we can add value. The short version: for an established, brownfield, AI-enabled team, the center of gravity is **Marshal** — a host-based engine that layers governance and cross-provider verification onto the setup you already run — plus the option to adopt individual capabilities à la carte. It runs on your engineer's own machine, inside your repository, on top of the harness your team already uses. Its state lives in a .marshal/ directory in the repo. No multi-tenant infrastructure, no standing connection into your systems, no runtime coupling. Best when you want to keep operating your own practice and add rigor to it.

With Marshal, you do not need to abandon what you've built. The point is to add the rigor that separates a working prototype from a system you can vouch for — on top of your stack, not in place of it.

This contrasts with **BLA's AI Factory** which is the BLA-run engine with these same capabilities plus all the agents, harnesses and integrations needed for a greenfield deployment. The Factory is suited for independent development where you are looking for output, not operation.

Core methodology

Both products share a single core we call **Cross-Provider Verified Development (CPVD)** — spec-driven development plus two structural invariants:

- **Cross-provider debate on load-bearing decisions.** A critic from a *different* model provider family attacks the proposal; the original agent rebuts; a human arbitrates the result before it is locked.
- **The executor cannot self-certify.** Verification comes from an independent model, not the one that wrote the code.

Why cross-*provider*, specifically — the point a sophisticated buyer will and should test: model families share training data and benchmark exposure, so they tend to

make *correlated* mistakes and sample the *same* blind spots. A second opinion from the same family is not a second prior. CPVD makes provider diversity a structural invariant rather than advice, so the reviewer that catches the confidently-wrong answer is the one least likely to share the author's blind spot.

Why Marshal fits an established practice

Marshal is built to layer onto a setup that already exists. Concretely:

- **It wraps your harness; it doesn't replace it.** Marshal drives the AI coding tool your engineers already use. It doesn't ask you to switch model, IDE, or workflow.
- **It respects your repository.** Marshal's discipline rules install as a single tracked import line in your project config; it never rewrites a config file you own, and on a managed or generated file it refuses to touch it and tells you so. Engine upgrades refresh locally with zero diff in your shared repo — your team never reviews churn they didn't author.
- **Zero runtime coupling.** Marshal introduces no external service, CLI, or runtime dependency into your stack. It reads your repo and drives your harness; it is never in your production path.
- **You own everything, in formats you can read without us.** Plans, decisions, gate proofs, and captured knowledge are plain markdown and JSON committed to your repository. If Marshal went away tomorrow, every artifact stays, and any competent engineer can read it.

This is the difference between buying a tool and inheriting a liability: Marshal's footprint in your codebase is designed to be small, legible, and reversible.

The capabilities: take the whole engine, or just the parts you need

This is the real offer for an established team. Each capability below is individually valuable and individually implementable into an existing pipeline. We implement them *with* your engineers, tuned to your stack and standards — you don't have to adopt Marshal wholesale to get any one of them.

Cross-provider review (the CPVD gate / Council Mode)

- **What it is:** on any load-bearing change — architecture, a security surface, scope, a dependency, a public API — a critic from a different provider family attacks the decision, the author rebuts, and a human arbitrates a recorded decision matrix.

- **The failure it prevents:** the confidently-wrong single-model call — the plausible architecture no one can vouch for, the migration that looks complete but dropped rows, the auth fix that quietly opens a new hole.
- **How it plugs in:** it wraps your existing review step and routes to whichever providers you already hold credentials for. The output is a recorded matrix a human signs, never a silent merge.

PR-gate enforcement

- **What it is:** no pull request opens except through a gate that runs your full verify bar, a secrets scan, and the cross-provider review — then stamps a commit-keyed proof into the PR body. A server-side branch-protection check refuses any PR whose stamp doesn't match its current head.
- **The failure it prevents:** the ungated merge — specification misses, secrets in the diff, and quality shortfalls reaching a client-visible branch; and the "I ran the checks" claim with no evidence behind it.
- **How it plugs in:** a local pre-push hook plus a CI status check wired to your existing branch protection (GitHub, Bitbucket, or equivalent). It is an enforcement wall, not a convention an engineer can forget.

Durable, isolated execution (inline or dispatched)

- **What it is:** work is planned as a durable spec → debate → plan → execution pipeline that survives session boundaries. Tasks run either *inline* (the engineer's agent does them in-session) or *dispatched* (headless and parallel), each in an isolated git worktree with a per-task verify and a fence that blocks a task from writing files it never declared.
- **The failure it prevents:** the plan that evaporates at the context limit; a task silently touching files outside its scope; parallel tasks colliding on the same code.
- **How it plugs in:** the durable plan is markdown in your repo; execution drives your existing harness. You choose inline for serial work and dispatch for parallel width.

Write-path round-trip evidence

- **What it is:** any data-mutating change ships with proof — seed, mutate, read back, and assert that the target field changed *and* the untouched fields survived.

- **The failure it prevents:** the silent-data-loss class — an edit that nulls an unedited column, a toggle that no-ops, a form missing a column the update writes.
- **How it plugs in:** a gate requirement layered onto your existing test suite.

Provider-diversity governance

- **What it is:** orchestration, bulk execution, and independent critique are deliberately assigned across providers; work routes around a provider outage or price change automatically; a self-hosted model option exists for regulated or sensitive data, so nothing leaves your hardware.
- **The failure it prevents:** single-vendor blind spots, and single-vendor lock-in and outage exposure.
- **How it plugs in:** a routing layer over the provider credentials you already hold.

The verification packet

- **What it is:** every delivery produces an audit-grade record — a scope-coverage mapping, quality evidence, a decision log, and a runbook — provable against the agreed scope by an independent reviewer.
- **The failure it prevents:** "the builder says it's done" as the only evidence of done.
- **How it plugs in:** it is generated from your engagement's own recorded event stream, and it reads cleanly for an auditor or a new team.

Where The Factory still fits

Even for a capable in-house team, there are shapes where the full BLA-run engine earns its place:

- **A bounded, high-risk migration or compliance rewrite** you would rather not staff off your own roadmap.
- **A surge** — a fixed-scope build you want delivered in parallel to your team's normal work.
- **A proof** — one delivery cycle, led by us, on a real product of yours, so you can watch the methodology produce working software and a verification packet before committing to running it yourself.

In each case you get the same CPVD methodology and the same verification packet. The only difference is who operates the engine.

How we'd start

Three entry points, smallest first — each stands on its own, and you can move between them:

1. **Adopt one capability.** Pick the gap that hurts most — for most established teams it's the cross-provider gate or PR-gate enforcement — and we implement it into your pipeline, working alongside your engineers.
2. **Run Marshal as your discipline spine.** Layer the whole engine onto one team's repository and harness, keeping your models, your IDE, and your workflow.
3. **Prove it with a Factory cycle.** One BLA-led delivery on a real product of yours, ending in working software plus an audit-grade verification packet.

You already built something real. We're here to make it something you can prove — at whatever level you choose to start.

Start a conversation

Tell us what you want to add to what you already run.
Email: support@ballastlane.com
Web: ballastlane.com
Phone: (781) 631-2536
Office: 1 Spring Street, Suite 1, Marblehead, MA 01945