Chapter 4: Research Proposal

Towards a Progressive Scalability for Modular Monolith Applications

Mauricio J. Carvalho dos Santos — ITA (WIP Draft)

February 2026

Contents

This chapter presents the core academic potential contribution of this research: a set of architectural guidelines designed to support software startups in adopting modular monolithic architectures that preserve scalability, maintainability, and internal modularity. These guidelines are grounded in the gaps and trade-offs identified through the systematic literature review and reflect the specific architectural tensions faced by early-stage software teams operating under conditions of limited resources, rapid iteration, and evolving requirements. Rather than offering a static framework, the guidelines aim to function as decision-making heuristics, designed to be actionable principles that inform architectural evolution without prescribing a single and rigid path.

The original research design identifies twelve guidelines organized across four analytical dimensions introduced in Chapter : Architectural Design, Operational Fit, Organizational Alignment, and Guideline Orientation. This dissertation fully develops the first six guidelines—G1 through G6—covering the Architectural Design and Operational Fit dimensions. These six guidelines form a complete, self-contained framework that addresses module boundaries, maintainability, progressive scalability, migration readiness, deployment strategy, and observability. The remaining six guidelines (G7–G12), spanning the Organizational Alignment and Guideline Orientation dimensions, are identified and scoped but deferred to future research (Section 4.10).

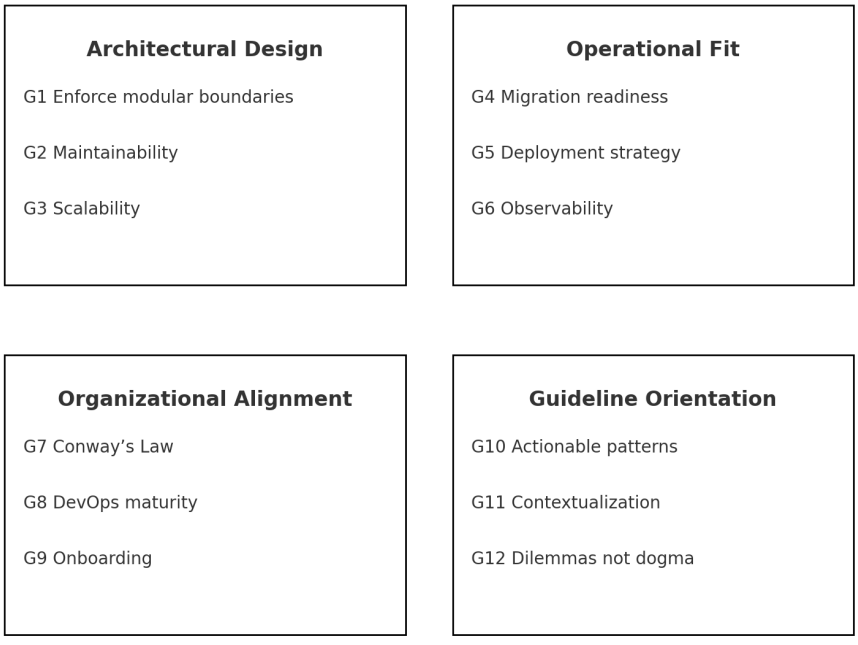


Figure 1: Four guideline dimensions for evaluating modular monolith architectures

Figure 1 shows the four guideline dimensions used in this research to guide the evaluation of modular monolith architectures in startup environments:

- **Architectural Design.** Covers criteria related to the software's internal structure.

Guideline G1 enforces clear module boundaries, G2 evaluates maintainability over time, and G3 assesses the system's scalability potential.

- **Operational Fit.** Addresses the operational requirements needed to support production enthronements. Guideline G4 checks migration readiness (for example, transitioning toward microservices), G5 focuses on deployment strategy and automation, and G6 considers observability levels (logging, metrics, monitoring).

- **Organizational Alignment.** Focuses on aligning the software architecture with the organization's structure. Guideline G7 examines team organization according to Conway's Law, G8 evaluates DevOps maturity, and G9 assesses how easily new team members can be onboarded without technical friction. These guidelines are identified and scoped in this dissertation but deferred to future research (Section 4.10).

- **Guideline Orientation.** Refers to high-level recommendations that steer design decisions without imposing rigid rules. Guideline G10 emphasizes practical, actionable patterns, G11 highlights the need to adapt recommendations to the specific business context, and G12 points out that trade-offs and dilemmas exist rather than unbreakable dogma. These guidelines are identified and scoped in this dissertation but deferred to future research (Section 4.10).

Each dimension group provides a set of guidelines that, together, form the foundation for the proposal of this research. It illustrates how technical, operational, organizational, and orientation-based considerations should be integrated when selecting or evolving a modular monolith architecture in software startups. What follows is a descriptive presentation of the guidelines within each dimension and their foundation concepts, starting with those that address the architectural design of modular monoliths.

4.1 Reference Implementation and Technology Stack

The guidelines proposed in this chapter are grounded in a reference implementation called **Tiny Store**[1], an Nx monorepo e-commerce application with four domain modules: orders, inventory, payments, and shipments. All code examples, exercises, and verification scenarios reference this implementation. Table 1 summarizes the production-grade technology stack used across the guidelines.

The deliberate choice of production-grade tooling from day one is central to the thesis argument: by embedding infrastructure such as Kafka, Temporal, and OpenTelemetry inside the monolith before any extraction occurs, the system is operationally ready for distribution while retaining the simplicity of a single deployable unit. These tools are not

[1] `https://github.com/maurcarvalho/tiny-store`

Table 1: Technology stack used in the Tiny Store reference implementation

Tool	**Purpose**	**Guid**
Nx	Monorepo management, dependency graph, affected CI	G1, G
ESLint + @nx/enforce-module-boundaries	Boundary enforcement at lint time	G1
KafkaJS + Apache Kafka	Event-driven communication, topic-per-aggregate	G4
Temporal	Durable workflows, saga orchestration, compensation	G4
Redis + ioredis	Caching layer (L1 vertical optimization)	G3
BullMQ	Async job queues (L2 async decoupling)	G3
TypeORM + PostgreSQL	ORM with per-module schema isolation	G3
OpenTelemetry	Distributed traces, metrics, structured logging	G6
Jaeger	Trace visualization and analysis	G6
Prometheus + Grafana	Metrics collection, dashboards, alerting	G6
Docker + Docker Compose	Local development, D0 baseline deployment	G5
Kamal	Production deployment, zero-downtime, SSL	G5
GitHub Actions	Module-aware CI with Nx affected	G5

migration prerequisites to be added later—they are architectural requirements that make progressive scalability achievable.

4.2 Background for Architectural Design Dimension

Modern software architecture treats modularity as a primary mechanism for balancing immediate development needs with long-term scalability. In the context of a modular monolith, this means structuring the application as a single deployable unit that internally consists of well-defined, self-contained modules. Each module acts as a building block of the system, encapsulating a cohesive set of related functionality behind a clear boundary. By imposing strict internal module boundaries and interfaces, a modular software design approach can preserve many benefits commonly associated with more distributed architectures, such as parallel development by multiple engineering teams, maintainability, and scalability, while avoiding a substantial portion of the operational complexity of distributed systems [ABGAZ2023DECOMPOSITION, GRZYBEK2020MODULAR]. In essence, the monolith's internals are made scalable and maintainable by design, and the system remains ready for gradual growth or future distribution, enabling progressive scalability [GRZYBEK2020MODULAR]. In summary, modular monolith architecture aims to achieve scalability and maintainability through disciplined modular boundaries within a single deployable unit.

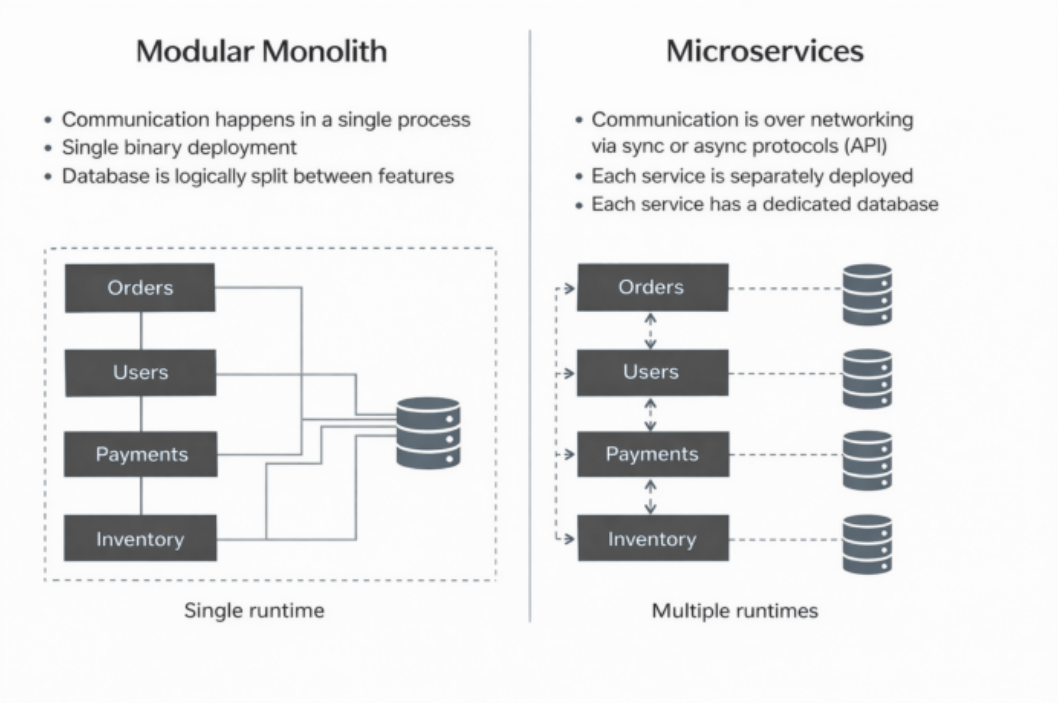


Figure 2: Deployment units in modular monoliths and microservices. The left side illustrates a modular monolith, where domain-aligned modules (e.g., Orders, Users, Payments, Inventory) execute within a single runtime and may share a single data store, while enforcing internal boundaries through code-level constraints. The right side illustrates microservices, where each service runs as an independent runtime and typically owns its persistence, enabling independent deployment and scaling at the cost of distributed-system operational overhead. Source: Dr Milan Milanović (adapted for this research).

Against this background, the discussion that follows places emphasis on the Architectural Design dimension. This research proposes that progressive scalability in modular monoliths depends first on enforceable modularity, meaning clear boundaries, explicit contracts, and build or test-time checks that detect boundary corruption early.

This priority also shapes the guideline order. The guidelines are presented by impact, with G1 as the foundation: without verifiable boundary enforcement, later guidelines are harder to apply because they assume stable module interfaces and controlled coupling. For that reason, this section introduces the concepts needed for the Architectural Design guidelines, including modular decomposition styles, coupling and cohesion, screaming architecture, layering, packaging strategies, and module design principles. The goal is to make explicit how coupling patterns affect the cost of change, and why domain-centric modularity helps sustain cohesion over time.

A useful way to frame architectural design is as an evolutionary progression in separation of concerns. Early architectural organization frequently optimized for technical separation, for example MVC and layered structures. Over time, the emphasis shifted toward dependency discipline and domain centric organization, as seen in Hexagonal Architecture, Onion Architecture, and Clean Architecture [COCKBURN2005HEXAGONAL,PALERMO2008ONION,MARTI

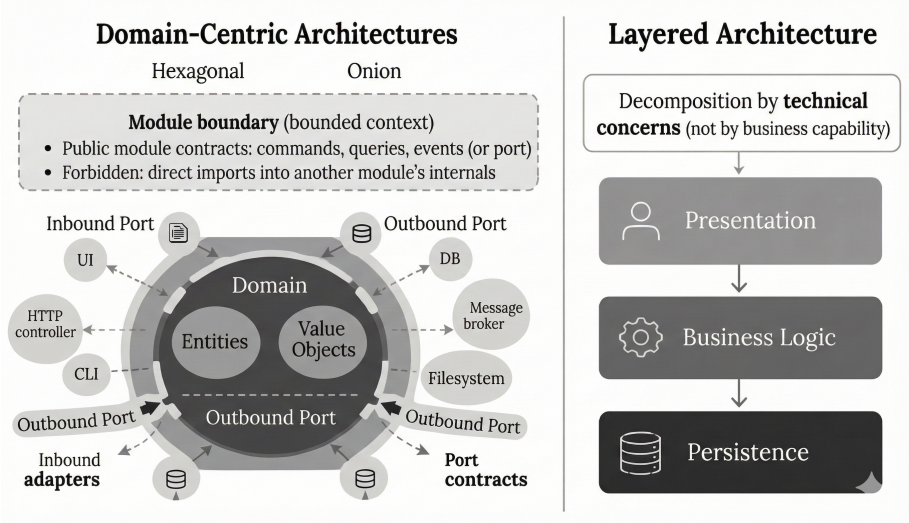


Figure 3: Layered versus domain-centric architecture. This representation supports the argument that domain-centric shifts the primary decomposition axis from technical layers to business capabilities, with the goal of improving cohesion and reducing cross-cutting coupling.

In parallel, packaging strategies evolved from technical layers to feature slices, and then to component-like modules that align more directly with domain boundaries [SHAW2018VERTICAL]. This research proposal positions modular monolith guidelines as a pragmatic state of practice that combines these influences, aiming to control change cost by constraining coupling and making boundaries verifiable in code.

Cost of Change: Coupling and Cohesion

Cost of change is often dominated by dependency structure rather than by local code complexity. As Russell L. Ackoff argues, *"A system is never the sum of its parts, it's the product of their interactions."* This perspective motivates treating coupling patterns as first order architectural concerns, because interactions are where change propagates, coordination increases, and unintended side effects emerge.

When introducing coupling and cohesion, this research proposal explicitly draws on the structured design foundations of Edward Yourdon and Larry Constantine, whose work provides a rigorous vocabulary for understanding how dependency structure shapes maintainability and change amplification [YOURDON1979STRUCTURED]. Their framing is used here to examine three claims that are central to the Architectural Design Dimension (G1–G3):

- **Not all coupling is born equal.** Some dependencies are stabilizing and intentional, particularly inside a coherent boundary, while others are accidental and create ripple effects across unrelated areas.
- **Decoupling has its cost.** Indirection, abstractions, and additional interfaces can reduce harmful dependency, but may increase cognitive load and implementation overhead if applied without need.
- **Cohesion is coupling in the right places.** Coupling is not eliminated, it is relocated into cohesive units so that change remains localized.

These claims motivate a baseline definition of what *"good modularity"* is aimed at in this research. Software design can be treated as the act of modeling units of **strong cohesion, loosely coupled** to each other. The Architectural Design Dimension (G1–G3) operationalizes this by proposing guidelines that make coupling explicit, constrain it through boundary rules, and preserve cohesion by aligning modules with business capabilities rather than technical utilities.

This framing also clarifies why domain centric modularity is emphasized. Functional decomposition is generally preferred over technical decomposition when the goal is to reduce change amplification, because it tends to localize business evolution inside a bounded unit. This directly relates to the distinction between intrinsic complexity, which comes from the domain and requirements, and accidental complexity, which comes from architectural and organizational choices. A recurring risk in modularization is mistaking organization for encapsulation, because directory structure alone does not enforce boundaries unless it is backed by interface contracts and dependency rules.

Modular Decomposition Styles: Horizontal vs Vertical

A fundamental design decision concerns how system functionality is decomposed. The two primary strategies are horizontal decomposition, organized by technical layer, and vertical decomposition, organized by feature or business capability. In traditional layered architectures, code is structured by technical responsibilities such as presentation, application services, and persistence. While this approach provides a form of separation of concerns, it often disperses domain logic across multiple layers and obscures the system's business intent [FOWLER2002PATTERNS]. As a result, understanding or modifying a specific feature frequently requires coordinated changes across several layers.

This change amplification effect can be stated directly. Changes to a layered architecture usually result in changes across all layers. The reason is that the dependency structure encourages features to cut across horizontal boundaries, even when the business change is conceptually localized. This coupling pattern increases the long term cost of change, because each new behavior requires cross layer coordination, additional integration effort, and broader regression risk.

Figure 4: Coupled-change ripple in a dependency graph. Nodes represent components and directed edges represent dependencies. The change originates at the top node (annotated as the starting point) and propagates along dependency paths. Darker nodes indicate components impacted by the initial modification, while lighter nodes remain unaffected. Concentric halos emphasize the spread of change through the system, illustrating how tighter coupling amplifies change impact beyond the original change.

Vertical decomposition addresses this limitation by organizing code around features, use cases, or domain capabilities. All elements required to implement a specific behavior, including API endpoints, business logic, and data access, are grouped within a single module or slice. This approach is commonly referred to as package by feature or vertical slice architecture [SHAW2018VERTICAL]. By aligning structural boundaries with business concepts, vertical decomposition tends to increase cohesion, reduce cross feature coupling, and make related code easier to find, because feature behavior is not scattered across horizontal layers.

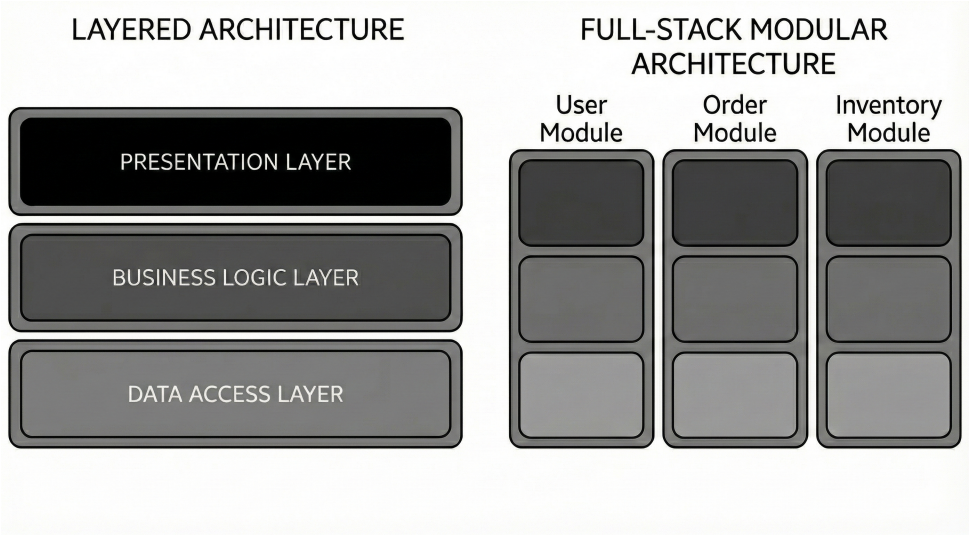


Figure 5: Layered and modular architecture. The left side shows a traditional layered decomposition, where code is organized by technical responsibility (presentation, business logic, and data access). The right side shows a modular decomposition, where domain modules (for example, User, Order, and Inventory) each contain their own UI/API, application logic, and persistence concerns, keeping feature changes more localized within module boundaries.

A key risk at this stage is architectural *"cargo culting"*, where teams adopt a packaging style or pattern without understanding the coupling problem it is intended to solve. It can also refer to the results of applying a design pattern or coding style blindly without understanding the reasons behind that design principle. In practice, this risk is elevated when architecture is documented as diagrams but not enforced in code, because teams can claim domain boundaries while the implementation continues to couple through shared models, utilities, or unrestricted imports.

Screaming Architecture: Domain Alignment and Model–Code Gap

A *screaming architecture* is one in which the system's structure communicates its business purpose at a glance. A well-designed codebase foregrounds domain concepts and use cases in its top-level organization, rather than privileging frameworks, infrastructure, or technical layering [MARTIN2012CLEAN,MARTIN2011SCREAMING]. Under this view, architecture is expressed less by diagrams and more by what the source code organization makes immediately visible, what the codebase "*screams*" becomes a proxy for architectural intent.

This perspective is particularly relevant to the Architectural Design Dimension (G1–G3), where modular boundaries are treated as first-class architectural elements. When the dominant structural signals emphasize technical concerns instead of domain concepts, the system tends to drift away from domain-centric modularity. Over time, this drift increases the cost of change, since engineers must reconstruct domain intent from dispersed artifacts, raising cognitive load and coordination overhead.

This risk connects to the *model–code gap* [FAIRBANKS2010JUSTENOUGH], the divergence between an architecture as documented and an architecture as implemented. Fairbanks argues that when architectural decisions are not made evident in code, they erode under delivery pressure, producing systems that appear compliant in documentation while contradicting the intended structure in practice.

To reduce this gap, Fairbanks proposes an *architecturally-evident coding style*, where architectural decisions are encoded directly into module boundaries and enforced dependency rules [FAIRBANKS2010JUSTENOUGH]. Mechanisms include explicit public interfaces, restricted import paths, and structural checks that fail when dependency constraints are violated. A central implication for this research is that modular boundaries should be explicit, inspectable, and continuously enforced by the codebase, making violations difficult to introduce unintentionally.

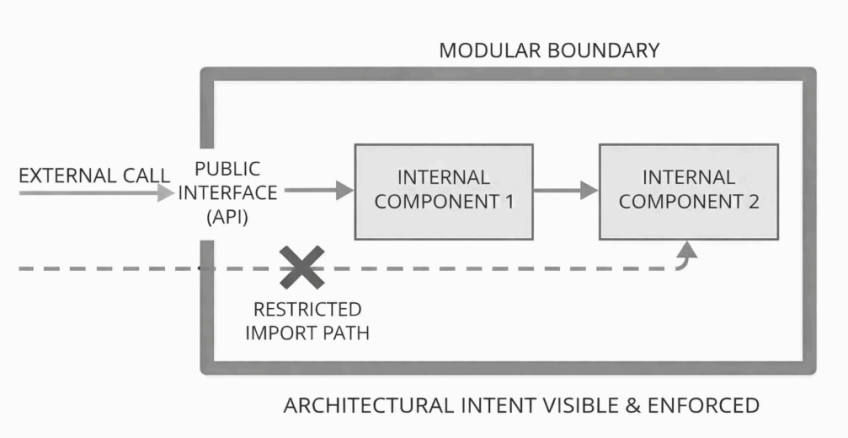


Figure 6: Explicit modular boundary with enforced imports.

An example of what a screaming architecture looks like in practice is illustrated by the high-level directory structure that will be explored in detail in the guideline examples. Even without inspecting individual files or previous knowledge of the codebase, the system's domain and capabilities are immediately apparent:

```
/modules/                  # Bounded contexts represented in Modules
    orders/
        domain/
            entities/          # Order aggregate
            value-objects/     # OrderItem, CustomerId
            events/            # Domain events
            repositories/      # OrderRepository (database operations)
        features/             # Use cases (vertical slices)
            place-order/           # Place Order services/use-cases
            cancel-order/          # Cancel Order services/use-cases
        listeners/             # Event handlers of Orders
    inventory/                 # Inventory module
    payments/                  # Payments module
    shipments/                 # Shipments module
```

At this level, the architecture already "*screams*" business capabilities such as orders, inventory, payments, and shipments, rather than technical layers or framework concerns. This structure makes domain boundaries, use cases, and integration points visible by construction, reducing the model–code gap before any implementation details are examined.

Within this research, screaming architecture and the model–code gap are treated as two facets of the same underlying problem. Architecture that exists only in documentation is fragile and prone to corruption. Architecture that is embedded in module boundaries, public APIs, and enforced dependency direction becomes continuously verifiable. This embedded nature is a prerequisite for controlling coupling, sustaining cohesion, and supporting progressive scalability within a modular monolith, and directly informs the Architectural Design Dimension (G1–G3) introduced in the following sections.

Separation of Concerns and Dependency Discipline

Separation of concerns can be implemented through architectural styles that constrain dependency direction. Layered Architecture typically structures dependencies outward to inward by technical responsibility, often represented as Presentation, Business, and Persistence layers [FOWLER2002PATTERNS]. While layering can improve local reasoning, it may also amplify change cost when features cut across layers and when domain logic is fragmented into anemic models and service orchestration.

Hexagonal Architecture, Onion Architecture, and Clean Architecture formalize a stricter dependency rule: business logic is insulated from delivery and infrastructure concerns, and dependencies must point inward toward stable domain abstractions [COCKBURN2005HEXAGONAL,PALE
A common representation of this structure is:

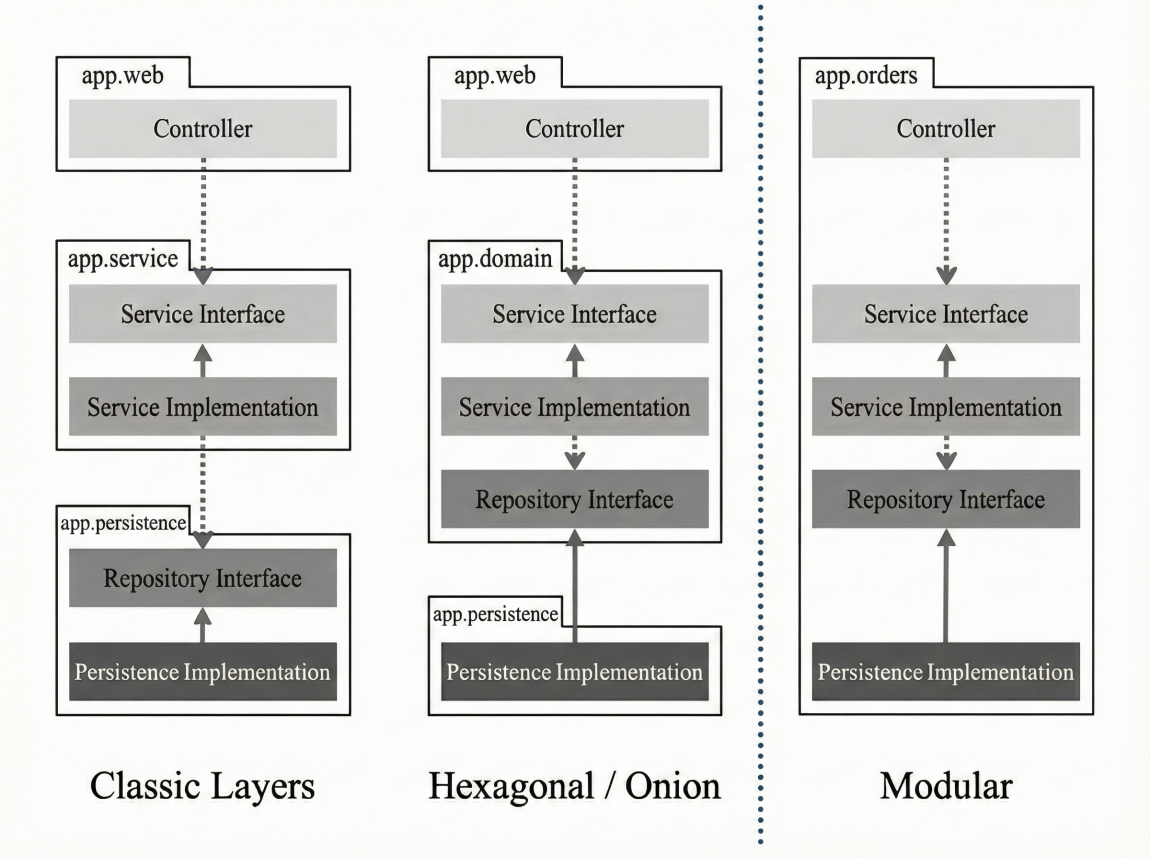


Figure 7: Comparison of architectural decomposition styles.

The intent is to reduce coupling to volatile infrastructure by inverting dependencies through interfaces and ports. This can reduce change cost because infrastructure replacements, testing strategies, and delivery mechanisms can evolve without rewriting core domain logic. However, dependency discipline alone does not guarantee domain centric modularity. A codebase can follow inward dependencies and still be organized as a single, tightly coupled domain blob if boundaries between business capabilities are not explicit.

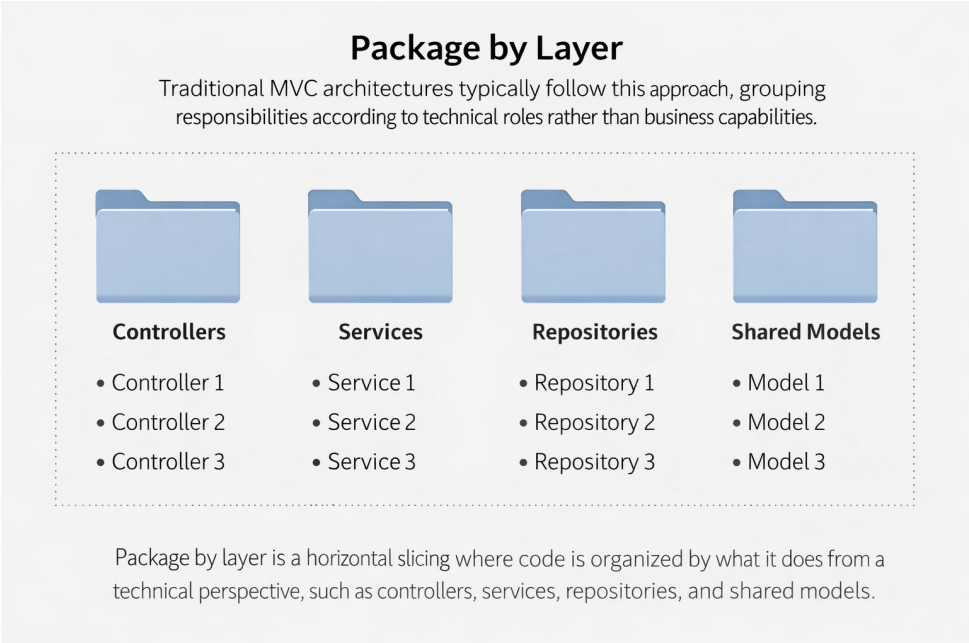
Packaging and Slicing Strategy

Packaging strategy determines what coupling is easy and what coupling is discouraged. This section compares three strategies using consistent terminology, because the Architectural Design Dimension (G1–G3) depends on understanding what each strategy optimizes for and where it tends to fail.

Package by layer is a horizontal slicing where code is organized by what it does from a technical perspective, such as controllers, services, repositories, and shared models. Traditional MVC architectures typically follow this approach, grouping responsibilities according to technical roles rather than business capabilities.

The primary failure mode is that business changes often span multiple technical packages, increasing coordination and regression risk. In layered architectures, even localized changes typically propagate across presentation, application, and persistence layers, amplifying the cost of change. This approach is closely associated with the model–code gap, since domain boundaries may be articulated in documentation while the code structure instead emphasizes technical layers.

This failure mode is reinforced by delivery-driven optimization, where short-term convenience encourages cross-layer shortcuts and shared abstractions. As business behavior evolves, conceptually narrow changes tend to trigger multi-layer modifications, increasing change amplification and obscuring architectural intent. Because domain boundaries remain implicit and unenforced in the code, the system gradually reflects technical decomposition rather than business capability, widening the model–code gap and weakening the sustainability of modular reasoning.

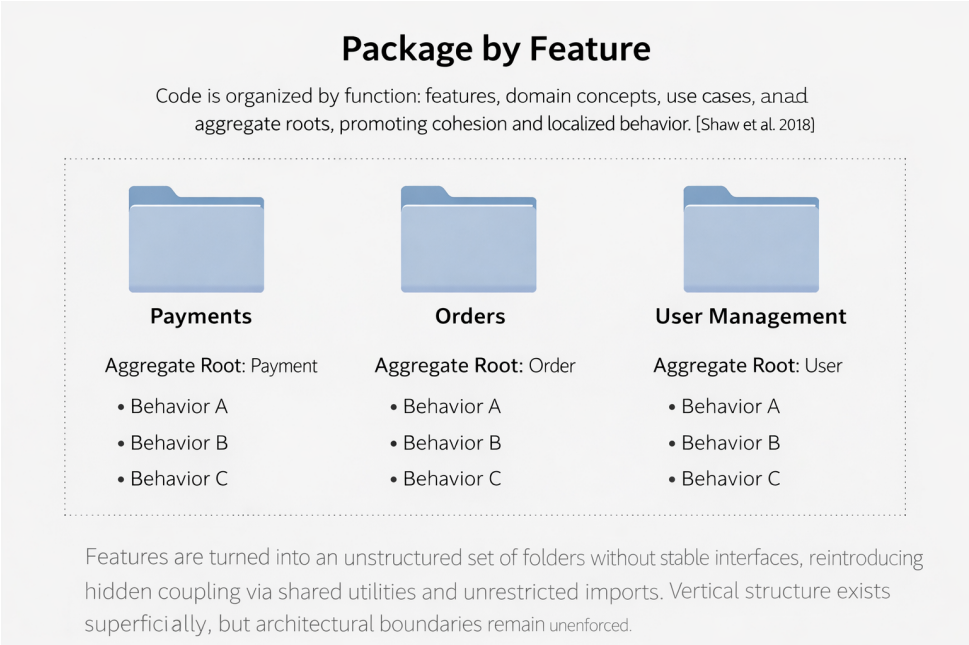


Package by feature is a vertical slicing where code is organized by what it does from a functional perspective, including features, domain concepts, use cases, and aggregate

roots. Cited benefits include higher cohesion, lower coupling, and related code being easier to find, because the implementation of a behavior is localized rather than scattered across technical packages [SHAW2018VERTICAL].

A common failure mode is turning features into an unstructured set of folders without stable interfaces, which can reintroduce hidden coupling through shared utilities and unrestricted imports. In such cases, vertical structure exists superficially, but architectural boundaries remain unenforced.

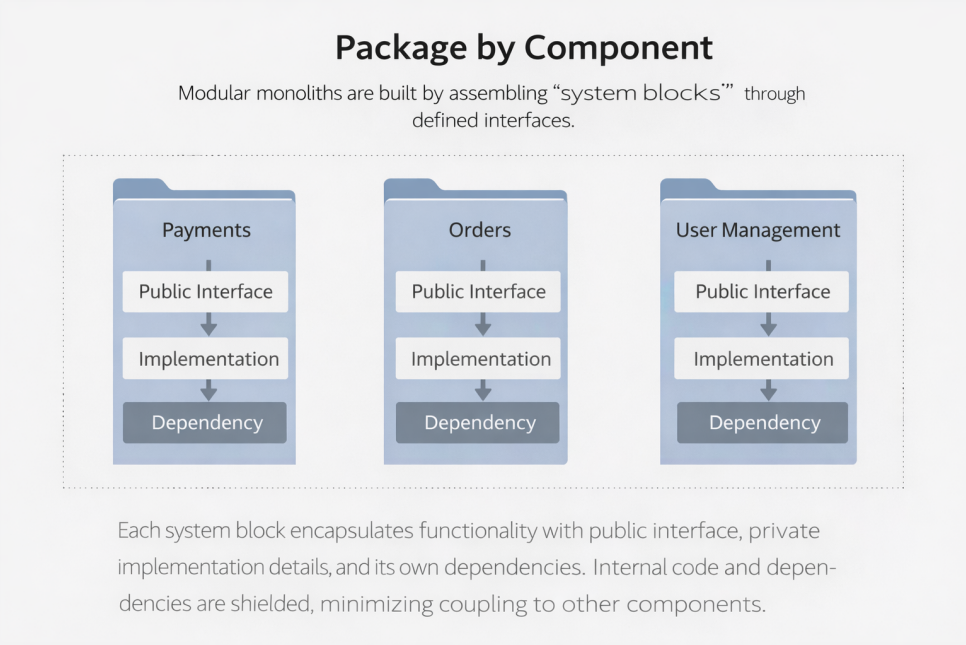
Another side-effect failure arises when nominal module boundaries are introduced without corresponding ownership of data and behavior. In these cases, modules expose internal state through shared data models, common persistence schemas, or cross-module service calls that bypass defined interfaces. Although the codebase may appear modular at the directory level, effective encapsulation is absent, allowing changes in one module to ripple unpredictably into others. This pattern undermines substitutability and independent evolution, and it often emerges when modularization is treated as a structural refactoring exercise rather than as an architectural commitment enforced through explicit contracts and dependency constraints.



Package by component is a packaging strategy in which code is organized around cohesive components, grouping functionality related to a specific responsibility. A component is defined as a unit of related behavior, accessed through a well-defined interface and encapsulated within the application boundary. This approach applies component-based or service-oriented design principles to a monolithic codebase, aiming to achieve modularity without introducing distribution.

The architectural hypothesis is that component boundaries become enforceable when they define what is exposed publicly, what is hidden internally, and which dependency directions are permitted between components. Under these conditions, package by component is the closest analogue to modules in a modular monolith architecture, where boundaries are enforceable constraints rather than organizational conventions, and coupling is intentionally limited.

A common failure mode arises when components are defined without a clear domain model, instead of being aligned with bounded contexts and aggregates as proposed in Domain-Driven Design [EVANS2003DDD]. Overly coarse components collapse multiple domain concepts into a single unit, accumulating unrelated responsibilities and reintroducing hidden coupling through shared models. Conversely, overly fine-grained components fragment aggregates and use cases across boundaries, increasing coordination overhead and weakening invariants that should be preserved within a single consistency boundary. In both cases, the absence of stable, intention-revealing interfaces grounded in explicit domain contracts undermines boundary enforcement, allowing dependencies to leak through shared abstractions or convenience-driven imports. As a result, components may exist as structural groupings without providing the isolation, autonomy, and scalability expected of modular units in a modular monolith.



Modularity as an Enforceable Principle

Modularity is treated at this research as an enforceable property rather than as a folder convention. Encapsulation and information hiding remain core mechanisms, because they minimize the number of potential dependencies by reducing what other modules can see and use [PARNAS1972CRITERIA]. This implies several practical principles that the Architectural Design Dimension (G1–G3) later operationalizes:

- Separating interface from implementation. Modules should expose a small public API and keep internal details private.
- Using encapsulation to minimize dependencies. Fewer public types and fewer cross module imports reduce change propagation paths.
- Matching public surface area to architectural intent. The surface area of internal public APIs and event listeners should reflect intended coupling and allowed integration points.
- Splitting the source code tree into multiple parts. Organizing the codebase into explicit module roots, for example /modules, helps architectural visibility and makes boundary enforcement feasible, because module ownership and dependency direction can be checked.

This view also reinforces the earlier distinction between organization and encapsulation. **Encapsulation over organization** means that directory structure matters only to the extent that it is backed by dependency rules, stable interfaces, and verification mechanisms. Without enforcement, modularity claims tend to collapse under delivery pressure, and coupling becomes implicit again.

Tiny Store's code repository [2] reflects the architectural hypothesis of modular boundaries and serves as a concrete artifact grounding the concepts introduced in this chapter. The project is publicly available and maintained as a reference implementation for this research, providing an auditable codebase in which packaging strategies and boundary enforcement mechanisms can be inspected directly.

The codebase is split into multiple parts that make architectural boundaries explicit, including a dedicated API layer and bounded context modules, with clearly defined locations for feature slices and for cross module integration through domain events. Although this chapter does not yet rely on detailed implementation examples, the repository is introduced here to ensure that the vocabulary and architectural constructs discussed in the Architectural Design Dimension (G1–G3) are mapped to concrete verifiable structures instead of remaining purely conceptual.

[2] `https://github.com/maurcarvalho/tiny-store`

Definition of Module Quality

The Architectural Design Dimension (G1–G3) treats module quality as a measurable architectural objective. Modules are therefore expected to satisfy the following properties:

- High cohesion and Low coupling
- Focused on a specific self contained business capability
- Aligned to a bounded context or aggregate
- Encapsulated data
- Substitutable
- Composable with other modules

This checklist is used later as the baseline for designing boundaries, for defining public APIs and events, and for evaluating whether a proposed module decomposition is likely to reduce change cost rather than merely reshuffle code.

Practical Evaluation for the Architectural Design Dimension (G1–G3)

This foundations chapter establishes a causal chain that the guideline templates and their verification plans will later operationalize. Coupling patterns dominate change cost, cohesion localizes change, separation of concerns reduces dependency on volatile infrastructure, and packaging strategy determines whether the code structure screams domain intent or technical layers. Modularity becomes sustainable only when interfaces, dependency direction, and enforcement mechanisms make boundaries explicit and verifiable.

In the next sections, the Architectural Design Dimension (G1–G3) will translate these foundations into guideline templates that specify intent, rationale, applicability conditions, supported by Tiny Store codebase as a reference artifact for reproducibility of examples and auditability convenience.

4.3 G1: Enforce Modular Boundaries

In this section, we present one of the concrete guidelines proposed for evolving a modular monolith application. G1 focuses on enforcing clear modular boundaries. The guideline aims to prevent unintended coupling in the codebase, preserve encapsulation, and ensure that dependencies between modules remain explicit and verifiable through build- and test-time checks.

Intent and Rationale

This research proposes that modularity in a monolith should be treated as an enforceable architectural property rather than an assumed outcome of refactoring. Without enforcement, boundaries tend to degrade over time through convenience-driven imports, shared utilities, and implicit wiring, which gradually reintroduces tight coupling. G1 therefore treats boundary enforcement as a preventive control, making coupling explicit, reviewable, and auditable, enabling independent evolution of modules and reducing the cost of architectural change.

Conceptual Overview

Define clear boundaries around each bounded-context module so that:

- Each module encapsulates its internal implementation and feature-level organization.
- Dependencies between modules occur only through explicitly declared and verifiable interfaces.
- Unintended coupling is prevented, enabling independent evolution of modules within a single deployable system.

Applicability Conditions and Scope

G1 is applicable when a system is decomposed into logical modules within a single codebase. Inter-module dependencies must be explicitly declared and automatically verifiable, independent of packaging, deployment, or distribution decisions. This guideline assumes a two-level decomposition strategy:

- *Bounded Context Modules:* The codebase is decomposed into bounded-context modules (for example, under a `modules/` root), and each bounded context is treated as a first-class module boundary.
- *Package-by-Feature Within Modules:* Inside each module, code is organized by feature (use cases, domain concepts, aggregates) to localize behavior and reduce scattering, without weakening the outer module boundary.

G1 focuses on enforcing the *outer* boundary between bounded-context modules. The internal package-by-feature organization is treated as a complementary structuring mechanism, and its detailed rules are addressed in subsequent topics.

Objectives

- *Clear Declaration and Isolation:* Ensure each module is clearly identified (for example, via naming conventions, package control, or a descriptor) and remains isolated unless an explicit dependency is declared.
- *Encapsulation:* Keep internal classes, data, and resources hidden within a module.
- *Controlled Dependency:* Allow one module to depend on another only when explicitly declared.
- *Visibility Management:* Declare which services, events, or commands a module exposes or consumes.
- *Early Verification:* Detect boundary violations at build or test time rather than at runtime.
- *Isolation Guarantees:* Optionally forbid certain modules from ever depending on each other.

Key Principles

- *Provided Interface vs. Internal Components:*
  - *Provided Interface:* Public classes, services, APIs, or events explicitly exposed to other modules.
  - *Internal Components:* Classes, utilities, or data that should remain private within the module.
- *Explicit Declarations:* All dependencies on other modules must be declared in a central descriptor to provide auto-verifiability (for example, annotation, configuration file, or YAML).
- *Forbidden Dependencies:* Declare modules that must never depend on each other to enforce stronger isolation where necessary.
- *Inter-Module Communication:* Use either synchronous calls via public interfaces or asynchronous event-driven channels, and avoid direct references to another module's internals.
- *Boundary-First Discipline:* Package-by-feature improves cohesion within a bounded context, but it does not replace module boundaries. A feature slice must not become a cross-module dependency mechanism (for example, by importing internal feature code from other modules).

Implementation Mechanisms

This research proposes implementing G1 through a combination of (i) structural encapsulation, (ii) explicit dependency declaration, and (iii) automated verification:

- *Structural encapsulation:* Separate public API packages from internal packages, and restrict visibility so that only the declared API is importable across module boundaries.
- *Declarative dependency model:* Maintain a module descriptor that defines which modules are required and which are forbidden, and keep it version-controlled with boundary changes.
- *Automated verification:* Fail the build when undeclared cross-module references are detected, and optionally verify event subscriptions and command usage against declared dependencies.

Metrics

- *Undeclared Dependency Reference Count:* Number of static references from module A to module B where $A \to B$ is not declared in the module descriptor.
- *Forbidden Dependency Reference Count:* Number of static references from module A to module B where $A \to B$ is explicitly forbidden.
- *API-Only Dependency Ratio:* For all cross-module references, the proportion that targets the provider module's declared public API surface (interfaces, exported packages, published events) rather than internal packages.
- *Encapsulation Leakage Count:* Number of external references to internal symbols (classes, packages, resources) that are not part of the provider module's declared public surface.
- *Module Isolation Test Pass Rate:* Percentage of module-scoped tests that execute with only the module and its declared dependencies available (for example, module-level context loading). Failures indicate hidden wiring to undeclared modules, even when imports look compliant.
- *Event Subscription Boundary Violations (if events are used):* Number of event handlers in module A consuming events from module B without a declared $A \to B$ dependency (or without a declared event subscription, depending on the enforcement model).
- *Boundary Bypass Surface Count (optional):* Number of occurrences of mechanisms that can bypass static boundaries (for example, reflection-based access, service locator patterns, dynamic class loading) within cross-module interaction paths. This metric is intended as a risk indicator even when no violations are detected statically.

Documentation Guidelines

- *Module Descriptor:* Maintain a machine-readable descriptor (YAML, annotation, or code) that centralizes *requires* and *forbids* entries for each module.
- *Change Log:* Whenever a boundary is added, removed, or modified, update the descriptor with version and author metadata.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to enforce modular boundaries should support:

- *Module Discovery:* Automatically identify modules via conventions (for example, module folders, annotations) or explicit configuration.
- *Boundary Verification:* Perform automated checks during build/test to detect undeclared cross-module references and fail on violations.
- *Event-Driven Enforcement:* Track published events and subscribers, enforcing that only modules with declared subscriptions handle specific events.
- *Isolated Testing:* Allow tests to load only a given module (and its declared dependencies), failing early if the module tries to wire code from undeclared modules.
- *Documentation Artifacts:* Generate up-to-date artifacts or reports for inclusion in architecture documentation or CI/CD feedback.
- *Runtime Validation (Optional):* Optionally perform runtime checks to catch dynamic boundary violations (for example, reflection) that compile-time checks might miss.

Literature Support Commentary

Although modularity is widely recognized as essential, much of the research treats it as an outcome of refactoring rather than a design criterion. Few works propose proactive structural mechanisms for enforcing modularity in monoliths, and those that do often lack empirical validation. This gap motivates G1 as a guideline that operationalizes modular boundaries through explicit declarations and automated verification.

4.4 G1 Applied: The Modular Tiny Store Example

This section operationalizes Guideline G1 (Enforce Modular Boundaries) using Tiny Store, a small modular monolith maintained as an executable reference for this dissertation.[3] The objective is to show how G1 is applied as an engineering practice, not only as an architectural principle. In Tiny Store, modularity is encoded in the repository structure, integration is wired in a visible composition root, and boundary erosion is surfaced as a failing check. The tutorial therefore connects the guideline to concrete actions: introduce a controlled violation, observe a crisp failure signal, trace it to an unauthorized dependency, and repair the system by restoring an explicit contract between bounded contexts.

The tutorial is written from the perspective of an engineer evolving a monolithic application that contains multiple business capabilities. The goal is to keep bounded contexts isolated, to ensure that any cross-module dependency is intentional and reviewable, and to prevent accidental coupling from accumulating into a refactoring bottleneck.

Two complementary feedback loops are used throughout the steps. The first is `npm run test:boundary`, which validates structural rules and fails when a module imports forbidden internals or bypasses the public surface of another context. The second is `npm run test:integration`, which exercises cross-module behavior end to end and confirms that the system remains correct after boundaries are tightened or contracts are introduced. Together, these targets support a workflow that is both practical and repeatable: make a change, observe the result immediately, and keep architectural integrity measurable as the system evolves.

Reader map

This tutorial is designed to be completed in a single sitting. It assumes you can run the repository's boundary and integration targets locally, and it keeps the steps intentionally hands-on. You will first trigger a boundary failure on demand, then locate the exact import or export that caused the violation, and finally fix it by introducing an explicit modular contract rather than importing another context's internals. The expected time investment is 20–30 minutes, and the outcome is a concrete, reproducible ability to detect and remediate boundary erosion using the same mechanisms that can be embedded into a continuous integration pipeline.

Repository Orientation

Tiny Store is the reference codebase used in this dissertation to ground architectural claims in an executable artifact. Its structure is designed to make modularity observable: boundaries are explicit in the directory layout, integration points are concentrated in

[3] `https://github.com/maurcarvalho/tiny-store`

a few predictable places, and the allowed dependency direction can be verified through automated checks.

The repository is an Nx monorepo: a single workspace that contains multiple projects (applications and libraries) managed under a shared toolchain and an inspectable dependency graph.[4] Deployable applications live under `apps/`, while reusable libraries live under `libs/`. This separation encodes a core rule of the modular monolith: composition roots are allowed to wire dependencies, while modules remain cohesive units that expose controlled integration surfaces.

The system's HTTP boundary is the `apps/api/` project, implemented with Next.js (App Router). REST endpoints under `apps/api/src/app/api/` act as thin adapters that translate requests into application operations. The same project also hosts shared runtime wiring under `apps/api/src/app/lib/`, so the API becomes the main composition root that assembles bounded contexts and connects infrastructure such as persistence and the event bus.

Cross-module integration is intentionally made visible. Listener subscription wiring is centralized in `apps/api/src/app/lib/register-listeners.ts`, which serves as an auditable map of event choreography: which events are subscribed to, which handlers react, and which dependencies exist at the system level. Centralizing subscriptions avoids hidden side effects (for example, listeners registered implicitly during import time) and makes integration decisions reviewable and testable.

Initially, domain behavior were divided into four bounded contexts under `libs/modules/`: `orders`, `inventory`, `payments`, and `shipments`. Each context is an Nx library structured around three concerns. The `domain/` subtree holds aggregates, entities, value objects, domain events, and repository abstractions. The `features/` subtree holds application use cases organized as vertical slices. The `listeners/` subtree reacts to published events and invokes local features, enabling collaboration without direct access to another context's internals.

Modules are consumed through a deliberate public API surface. Each bounded context exposes an entrypoint, typically `modules/<context>/src/index.ts`. This entrypoint is the module's contract: it re-exports only approved symbols (handlers, listener factories, public types, event contracts) and keeps domain internals private. As a result, other projects import capabilities through stable module-level paths (for example, `@tiny-store/modules-orders`) while boundary discipline remains enforceable.

The project makes event-driven collaboration the primary cross-module trace. Instead of "class-to-class" dependencies, the relevant path is "event → listener → feature." Orders emits domain events, Inventory reacts to reserve stock, and subsequent events drive Payments and Shipments. This choreography is expressed in each context's `domain/events/`, implemented by `listeners/`, and wired in the composition root via

[4] `https://nx.dev/concepts/monorepos`

`register-listeners.ts`.

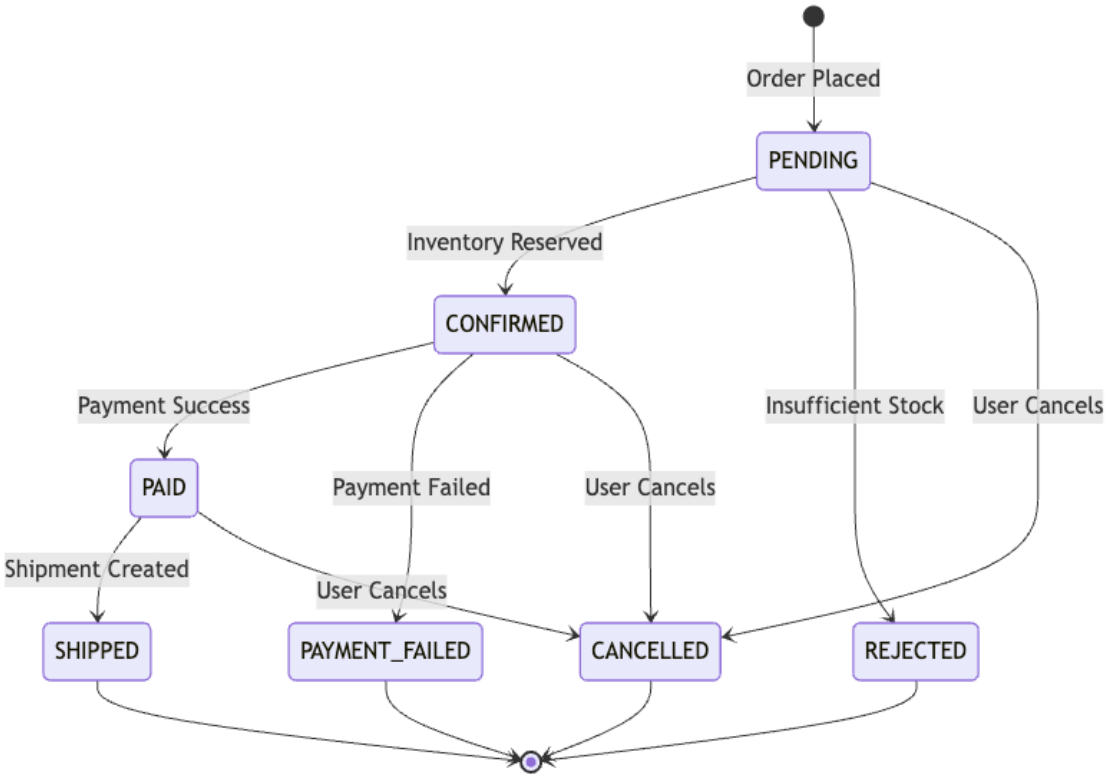


Figure 8: Lifecycle of order-related events across bounded contexts.

Statuses description:

`PENDING`	Order created, awaiting inventory check
`CONFIRMED`	Inventory reserved, ready for payment
`REJECTED`	Insufficient stock
`PAID`	Payment successful
`PAYMENT_FAILED`	Payment declined
`SHIPPED`	Shipment created and dispatched
`CANCELLED`	User cancelled order

Shared code is intentionally constrained under `libs/shared/`. The `domain/` library provides domain-neutral primitives, while `infrastructure/` provides reusable mechanisms such as the in-memory event bus and persistence utilities. This split reduces accidental coupling: domain meaning stays inside bounded contexts, and shared libraries remain a small set of stable building blocks rather than a catch-all dependency hub.

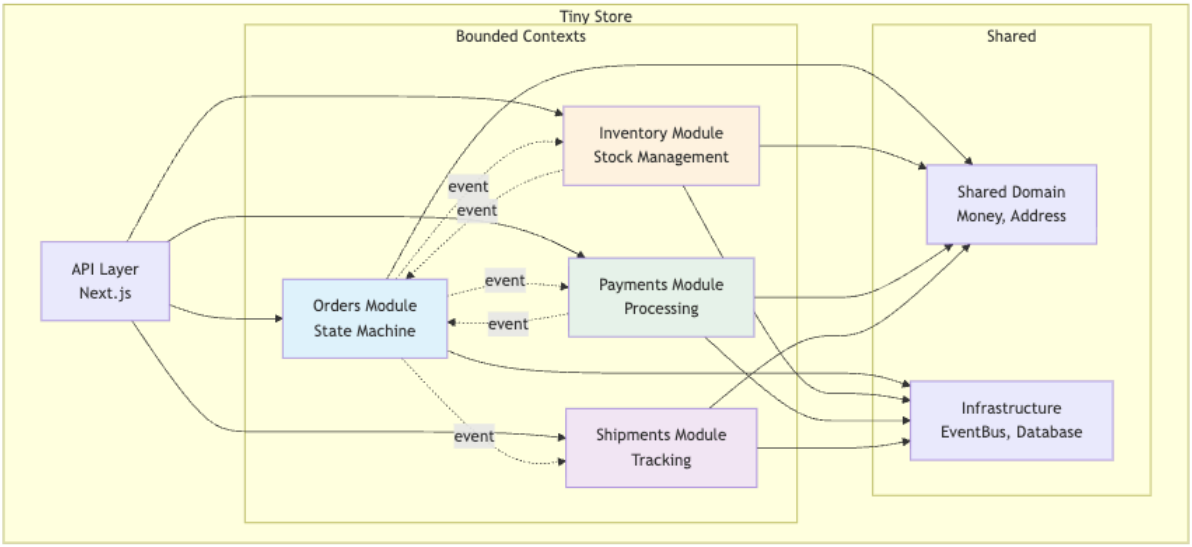


Figure 9: Repository-level orientation of the Tiny Store monorepo.

Architecture is enforced as executable checks. The `npm run test:boundary` target validates module dependency rules and detects coupling regressions, while `npm run test:integration` exercises cross-module flows end to end. Together, they provide fast feedback on both structural correctness and runtime behavior.

A practical reading order follows the enforced dependency direction: begin at `apps/api/` to see system assembly, then inspect `register-listeners.ts` to identify cross-context collaboration points. Next, enter a bounded context through its `src/index.ts` public surface and trace inward from `features/` into `domain/`. When behavior crosses contexts, follow the published events into `listeners/` and then into the next context's `features/`. This navigation strategy reflects the architectural intent: explicit contracts, visible integration, and boundaries that remain testable as the system scales.

This repository orientation establishes the concrete elements that G1 relies on: bounded contexts exposed through explicit entrypoints, cross-context collaboration expressed as event-driven contracts, and integration wiring centralized in the composition root. With this map in place, the tutorial can move from structure to action. The steps below apply G1 using Tiny Store as an executable reference. Each step states the intent and the expected signal, and when applicable it includes a runnable snippet that reproduces a boundary failure, pinpoints the violating dependency, and validates the fix through the same automated checks that enforce modular boundaries during day-to-day development.

Tutorial: Step-by-Step Application

The steps below apply G1 using Tiny Store as an executable reference. Each step states the intent and the expected signal, and when applicable it includes a runnable snippet.

Steps

- *Step 0: Establish a Clean Baseline:* Confirm that boundary enforcement is active before introducing controlled violations. Run the baseline checks, and verify that all targets pass.

Listing 1: Baseline checks for G1

```
npm install
npm test
npm run test:boundary
npm run test:integration
```

- *Step 1: Confirm Module Discovery is Unambiguous:* Verify that bounded contexts are mechanically discoverable by path conventions, enabling static boundary checks. In Tiny Store, bounded contexts live under `libs/modules/<context>/`, and shared primitives live under `libs/shared/`.
  - ✓ Do: treat `libs/modules/<context>/` as the outer boundary for G1.
  - ✗ Don't: create ad-hoc bounded contexts inside `libs/shared/` without enforcement.
- *Step 2: Treat Package-by-Feature as Internal Organization Only:* Confirm that feature slices improve cohesion inside a module while the outer boundary remains contract-based. Feature code (for example `src/features/`) may depend on `domain/` within the same module, but must not import other modules' features or domain internals.
  - ✓ Do: allow `features/` → `domain/` dependencies inside the same module.
  - ✗ Don't: allow module *A* feature code to import module *B* feature or domain internals.

- *Step 3: Separate Public Surface from Internal Components:* Ensure cross-module usage occurs only through module entrypoints (for example `@tiny-store/modules-orders`), so internals cannot leak by import convenience.

Listing 2: Public entrypoint import (allowed)

```
/**
 * PASS: DO: import the module via its public entrypoint.
 * The public surface should expose handlers, ports, and event
   contracts,
 * not domain entities or repositories.
 */
import { PlaceOrderHandler } from '@tiny-store/modules-orders';
```

The example above is correct because it imports a capability that the Orders module explicitly exposes as part of its contract. The rest of the system depends on what Orders *does* (a use case handler), not on how Orders *implements* its domain model. This is the preferred dependency shape: module-to-module dependencies point to stable entrypoints, and the set of exposed symbols remains small enough to be audited as the system grows.

Listing 3: Cross-module import of internals (blocked)

```
/**
 * FAIL: DON'T: entities and repositories should remain internal to
   the module.
 */
import { Order } from '@tiny-store/modules-orders';
// internal entity
import { OrderRepository } from '@tiny-store/modules-orders';
// internal repository
```

The code above is incorrect because it couples other contexts to Orders' implementation details. Other modules should not construct, persist, or reason about Orders via internal entities or repositories, even if the import path appears "official." Reactions to order state changes should flow through event contracts and listeners, while requests to Orders should use a use case handler exported by the public surface. When functionality is genuinely cross-cutting, promote it to `libs/shared/` as a domain-neutral primitive instead of importing or duplicating Orders' internals. This preserves module autonomy, keeps invariants local, and prevents boundary checks from being eroded by convenience imports.

- *Step 4: Encode Explicit Declarations (Descriptor Equivalent):* Keep dependency rules explicit and reviewable by expressing *requires* and *forbids* in a single, executable source of truth (in Tiny Store, the boundary test suite plays this role).

Listing 4: Example code-based boundary descriptor (illustrative)

```
/**
 * Example only: keep a single source of truth for module boundaries.
 * This can live near libs/shared/testing/src/module-boundaries.ts
 * and be imported by module-boundary.spec.ts.
 */
export const moduleBoundaries = {
  orders: {
    requires: ['shared-domain', 'shared-infrastructure'],
    forbids: ['inventory', 'payments', 'shipments'],
  },
  inventory: {
    requires: ['shared-domain', 'shared-infrastructure'],
    forbids: ['orders', 'payments', 'shipments'],
  },
} as const;
```

- *Step 5: Centralize Cross-Module Wiring in the Composition Root:* Ensure cross-module collaboration is auditable by keeping listener subscription wiring in the API composition root, rather than inside module internals.

Listing 5: Centralized wiring in the composition root (`apps/api/src/app/lib/register-listeners.ts`)

```
/**
 * PASS: DO: keep subscriptions centralized in the composition root
 * (apps/api/src/app/lib/register-listeners.ts) so cross-module
   wiring is auditable.

 * The sub binds an event name (contract) to a handler (reaction).
 */
import { EventBus } from '@tiny-store/shared-infrastructure';
import { OrderPlacedListener } from '@tiny-store/modules-inventory';

const eventBus = EventBus.getInstance();
eventBus.subscribe('OrderPlaced', (event) => {
  return new OrderPlacedListener().handle(event);
});
```

Listing 6: Hidden wiring inside a module (avoid)

```
/**
 * FAIL: DON'T: wiring inside a module makes dependencies implicit
   and harder to audit.
 * Subscriptions should be registered in the composition root
   instead.
 */
EventBus.getInstance().subscribe('OrderPlaced', handler);
```

- *Step 6: Enforce Boundary Verification as a First-Class Gate:* Boundary rules only protect the architecture if they run continuously. This step treats boundary verification as a mandatory gate in the local workflow and in CI, so boundary erosion is detected before it reaches runtime. In Tiny Store, the gate is `npm run test:boundary`: it encodes the allowed dependency direction between domains and fails the build whenever a module bypasses a public surface, imports forbidden internals, or introduces an unauthorized cross-context dependency. The key idea is to make architectural integrity measurable and non-optional, using the same feedback loop that already exists for unit and integration tests.

Listing 7: Boundary verification gate

```
npm run test:boundary

FAIL: Boundary violation detected

Source project:   libs/modules/orders
Forbidden import: libs/modules/inventory/src/domain/Product.ts
Importing file:
   libs/modules/orders/src/features/place-order/use-case.ts

Rule: Modules must import other modules only via their public
   entrypoint
      (e.g., @tiny-store/modules-inventory), not internal domain
   files.

Fix:  Replace internal import with an explicit contract:
      - consume a public handler, or
      - react via an event + listener, or
      - move truly shared primitives to libs/shared/.
```

The expected signal is binary: the gate either passes (no violations) or fails with a report that pinpoints the violating dependency.

In addition to the tooling-level gate, it is useful to have a focused automated test that asserts the intended public surface. The objective is not to "test architecture" by runtime imports alone, but to provide a readable specification of what must and must not be available through each module's entry-point. The example below checks that internal entities are not exposed, while public handlers are available. This test is complementary to `test:boundary`: it documents expectations and protects against accidental re-exports in `src/index.ts`.

Listing 8: Public surface test: internals are not exported, handlers are

```
/**
 * libs/shared/testing/src/module-boundary.spec.ts
 *
 * This test asserts the module public surface (entrypoint exports).
 * It complements test:boundary by preventing accidental re-exports
    of internals.
 */
describe('Module Public Surface', () => {
  it('does not expose Inventory internals via the module
    entrypoint', async () => {
    const inventory = await import('@tiny-store/modules-inventory');

    // Internal domain types must NOT be part of the public API
    surface.
    expect((inventory as any).Product).toBeUndefined();
    expect((inventory as any).InventoryRepository).toBeUndefined();
  });

  it('exposes Orders public handlers via the module entrypoint',
    async () => {
    const orders = await import('@tiny-store/modules-orders');

    // Public capabilities are allowed and expected.
    expect((orders as any).PlaceOrderHandler).toBeDefined();
    expect((orders as any).GetOrderHandler).toBeDefined();
  });
});
```

When this gate is treated as first-class, the architectural rules stop being informal conventions and become enforced constraints. In practice, the boundary policy

can be summarized as follows: external modules must not access another context's entities, repositories, or internal services; they may depend on public handlers, event contracts, and listener wiring through the composition root; and they may reuse domain-neutral primitives from `libs/shared/`. This is the mechanism that keeps G1 enforceable as the codebase scales.

- *Step 7: Add Module Isolation Checks to Detect Hidden Wiring:* Detect coupling that does not appear as imports by running module-scoped isolation tests that load a module with only its declared dependencies available.

Listing 9: Illustrative module isolation test

```
/**
 * Example only: the test should fail if the Orders module tries to
 * access undeclared modules during wiring or handler execution.
 */
describe('Orders module isolation', () => {
  it('loads with declared dependencies only', async () => {
    const module = await loadModule('orders', {
      allowed: ['shared-domain', 'shared-infrastructure'],
    });
    expect(module).toBeDefined();
  });
});
```

Exercise Walkthrough: Controlled Violations and Signals

The exercises below intentionally break modular boundaries to demonstrate that enforcement is active and to make failure signals actionable. Each exercise introduces a single, controlled breach, runs the same automated gates used in day-to-day development, and observes the resulting signal. This structure serves two purposes: it confirms that the boundary rules are not merely documented, and it trains the reader to diagnose violations quickly by linking a concrete change to a predictable failure mode.

To keep the summary compact, the table uses a lightweight metric notation. $C_\bullet$ denotes a *count* of violations or disallowed references (higher is worse), while $P_\bullet$ denotes a *pass rate* (higher is better). In particular, C_{leak} counts public-surface leaks (internal symbols exposed or consumed externally), C_{undecl} counts undeclared cross-module references, C_{forbid} counts references that violate an explicit forbidden dependency rule, P_{iso} captures the isolation pass rate when module-scoped tests are executed under declared dependencies only, and C_{event} counts boundary violations in event subscription and handling.

Exercise summary

Ex.	What you change	Run	Signal / metric
0	Establish baseline	`test:boundary` `test:integration`	pass (baseline)
1	Export internal symbols (leak)	`test:boundary`	fail, $C_{\text{leak}} \uparrow$
2	Add direct cross-module import	`test:boundary`	fail (or add rule), $C_{\text{undecl}} \uparrow$ / $C_{\text{forbid}} \uparrow$
3	Subscribe inside module (hidden wiring)	`test:boundary` `test:integration`	fail (rule/iso), $P_{\text{iso}} \downarrow$ / $C_{\text{event}} \uparrow$

Exercises

- *Exercise 0: Confirm the Baseline:* Re-run Step 0, and confirm that all checks pass.
- *Exercise 1: Create an Encapsulation Leak by Exporting an Internal Symbol:* Demonstrate that entities and repositories must not be exposed through module entry-points. Introduce the violation by exporting internal symbols from the Orders module entry-point (for example `libs/modules/orders/src/index.ts`).

Listing 10: Violation: exporting internal symbols

```
/**
 * FAIL: VIOLATION: exporting internal domain symbols leaks
   encapsulation.
 */
export * from './domain/entities/order';
export * from './domain/repositories/order-repository';
```

Run enforcement.

Listing 11: Run boundary enforcement after Exercise 1

```
npm run test:boundary
```

Expected signal: a boundary test fails, and $C_{\text{leak}} \uparrow$. Fix: remove these exports, and expose a contract instead (handler/port/event).

- *Exercise 2: Introduce a Direct Cross-Module Import (Undeclared or Forbidden):* Demonstrate that bounded contexts must not couple via direct imports for convenience. Introduce the violation by adding a direct import in an Orders feature file under `libs/modules/orders/src/features/`.

Listing 12: Violation: direct cross-module import

```
/**
 * FAIL: VIOLATION: Orders directly imports Inventory.
```

```
 */
import { SomeInventoryHandler } from '@tiny-store/modules-inventory';
```

Run enforcement.

Listing 13: Run boundary enforcement after Exercise 2

```
npm run test:boundary
```

✗ Expected signal: a boundary check fails if cross-module imports are constrained by the enforcement suite. If it does not fail, treat this as an enforcement coverage gap and extend `module-boundary.spec.ts` to detect forbidden `@tiny-store/modules-*` imports from within another module. Metric impact: $C_{\text{undecl}} \uparrow$ or $C_{\text{forbid}} \uparrow$. Fix (preferred): replace the direct import with event-driven collaboration (publish `OrderPlaced`, react in Inventory, publish `InventoryReserved`).

- *Exercise 3: Hide Event Wiring Inside a Module (Implicit Dependency):* Demonstrate why subscription wiring must remain centralized and auditable. Introduce the violation by placing a subscription call inside a module file (for example under `libs/modules/<context>/src/listeners/`).

Listing 14: Violation: hidden event wiring

```
/**
 * FAIL: VIOLATION: hidden event wiring inside a module.
 */
EventBus.getInstance().subscribe('OrderPlaced', (e) =>
   this.handle(e));
```

Run enforcement.

Listing 15: Run enforcement after Exercise 3

```
npm run test:boundary && npm run test:integration
```

Expected signal: this may not fail immediately if the current enforcement suite does not scan for subscription calls outside the composition root. In that case, enforce a rule that forbids `eventBus.subscribe(` usage outside `apps/api/src/app/lib/register-listen` and/or detect it via module isolation tests. Metric impact: $P_{\text{iso}} \downarrow$ and potentially $C_{\text{event}} \uparrow$. Fix: move the subscription back to `register-listeners.ts`.

Conclusion of the G1 Implementation

This section demonstrated that enforcing modular boundaries in a modular monolith is not a matter of documentation or developer discipline alone. In Tiny Store, G1 is implemented as an executable practice: bounded contexts are exposed through explicit entry-points, cross-context collaboration is expressed through contracts (events, handlers, and listener registration), and integration wiring is kept visible in the composition root. The result is that coupling becomes observable and auditable. When a boundary is violated, the failure signal is immediate and actionable, pointing to the exact dependency that must be removed or formalized.

The walkthrough exercises reinforced the practical distinction between the right and wrong dependency shapes. The right shape is contract-based: modules depend on other modules through public handlers and event contracts, and subscriptions are registered centrally so that integration is reviewable. The wrong shape is convenience-based: importing another module's internals, leaking symbols through entry-points, or hiding wiring inside modules. These shortcuts may appear productive in the short term, but they create architectural debt by turning internal implementation details into system-wide dependencies.

Most importantly, the implementation showed how G1 remains enforceable over time. The boundary verification gate (`npm run test:boundary`), complemented by integration checks (`npm run test:integration`) and public-surface assertions, turns modularity into a measurable constraint. This makes boundary erosion difficult to introduce accidentally and inexpensive to fix when it occurs. As Tiny Store evolves, the same mechanisms scale with it: new modules can be added, contracts can be expanded deliberately, and the architecture can be kept stable without slowing down delivery. In this sense, G1 is not only a guideline but also the foundation that enables progressive scalability by keeping decomposition options open while the system remains a single deployable unit.

With boundaries enforced and coupling kept explicit, the next constraint shifts from *whether* modules can remain isolated to *how* they can be evolved safely. This transition motivates Guideline G2, which focuses on maintainability: preserving clarity of change, controlling complexity growth, and ensuring that modifications remain localized, testable, and low-risk as the codebase and team scale.

4.5 G2: Embed Maintainability

This section presents G2, which focuses on embedding maintainability as an architectural property of modular monolith applications. In this dissertation, maintainability is framed economically as a bounded cost of change: the architecture constrains the expected effort, risk, and coordination required to modify the system as it evolves. Rather than treating maintainability as an indirect outcome of refactoring or developer discipline, G2 operationalizes it through explicit design constraints, stable interaction contracts, and continuous detection of architectural drift.

G2 builds directly on G1. While G1 enforces the existence of explicit modular boundaries, G2 ensures that those boundaries remain effective over time. Together, these guidelines prevent the gradual erosion that often causes traditional monoliths to regress into tightly coupled systems with monolithic change dynamics.

Intent and Rationale

Empirical studies consistently show that the primary driver of maintainability degradation in monolithic systems is not system size, but the accumulation of hidden coupling, ambiguous dependency direction, and architectural drift over time [GRAVANIS2021DONT,BLINOWSKI2022MONO
As systems evolve, convenience driven shortcuts such as deep imports, shared utilities, and implicit wiring gradually increase the blast radius of change and raise coordination costs.

This dissertation treats maintainability as a preventive architectural concern. G2 preserves bounded cost of change by constraining how modules interact, how dependencies are introduced, and how responsibilities are distributed. The guideline prioritizes change locality, stable contracts, and controlled dependency growth over short term reuse. These properties ensure that most changes remain confined to a single bounded context and that the effort required to evolve the system scales sublinearly with its size.

G2 extends G1 from boundary correctness to boundary sustainability. While G1 makes dependencies explicit and verifiable, G2 ensures that those dependencies remain limited in scope, stable under evolution, measurable trough specific metrics and economically manageable as the system grows [ARYA2024BEYOND,BERRY2024ISITWORTH].

Conceptual Overview

Maintainability is embedded by designing modules so that change remains local and predictable:

- Each bounded context exposes a narrow and intentional public surface that represents stable capabilities rather than internal structure.
- Cross module interaction occurs exclusively through explicit contracts.
- Dependency direction reflects responsibility and information flow, reducing the risk of cycles and cascading change.
- Architectural drift is detected early through executable and observable structural signals.

Applicability Conditions and Scope

G2 applies to systems organized as modular monoliths, where multiple bounded context modules coexist within a single codebase and are expected to evolve continuously. The guideline assumes that:

- Module boundaries are explicitly defined and at least partially enforced, as established by G1.
- Each module has clear ownership and a well defined responsibility.
- Architectural drift is a realistic risk during ongoing feature development.

G2 does not prescribe specific frameworks, build tools, or CI pipelines. Its scope is limited to metrics definition, design time and code level practices that directly influence long term changeability.

Objectives

- Bound the cost of change by limiting change propagation across modules.
- Stabilize dependency direction through disciplined layering and contract first boundaries.
- Constrain the growth of module public surfaces.
- Detect architectural drift before it becomes structurally expensive to reverse.
- Preserve the option for incremental refactoring and future decomposition.

Key Principles

- *Encapsulation before reuse:* Cross-module reuse of internal code is treated as a maintainability risk unless mediated through an explicit contract.
- *Contract over structure:* Modules depend on declared interfaces or events, not on internal implementation details.
- *Unidirectional dependency flow:* Dependency direction reflects responsibility and information flow, and cycles are considered maintainability violations.
- *Change locality:* Most changes should be implementable within a single module.
- *Evolution over speculation:* Abstractions are introduced to support observed change, not anticipated reuse.

Implementation Mechanisms

G2 is implemented through established software engineering practices rather than framework-specific mechanisms:

- *Layered internal structure:* Within each bounded context, code is organized into domain, application, and infrastructure concerns, ensuring that business rules remain insulated from volatile technical details.
- *Explicit module entrypoints:* Each module exposes a single public surface that defines its externally visible capabilities.
- *Contract-oriented collaboration:* Cross-module interaction occurs through synchronous interfaces or published domain events.
- *Centralized composition:* Wiring between modules is visible and auditable, preventing implicit coupling.
- *Governed exceptions:* Deviations from dependency or layering rules require explicit justification.

Common Failure Modes and Anti-Patterns

The following failure modes are frequently observed in modular monoliths that lack explicit maintainability discipline. Each anti-pattern increases the cost of change by expanding the blast radius of modifications, weakening ownership boundaries, or introducing hidden coupling. For each anti-pattern, G2 defines one or more metrics that make the degradation observable and verifiable.

- *Cross-Module Internal Reuse:* Reusing internal classes or utilities from another module instead of interacting through a declared public API tightly couples the consumer to the provider's internal structure. Internal refactoring then forces coordinated changes across modules, undermining independent evolution. This anti-pattern is primarily detected through an increase in encapsulation leakage count

(C_{leak}) and a reduction in the API-only dependency ratio (ρ_{api}).

- *Implicit Dependency Introduction:* Dependencies introduced through configuration, dependency injection wiring, or reflection without explicit declaration bypass architectural review. Over time, they erode the reliability of dependency models and increase the effort required to assess change impact. This failure mode is reflected by an increase in undeclared dependency references (C_{undecl}) and by reduced module isolation test pass rates (P_{iso}).
- *Bidirectional or Cyclic Dependencies:* Allowing modules to depend on each other directly or indirectly introduces cycles that prevent independent evolution. Cycles increase cognitive load and coordination cost and are expensive to remove once established. This anti-pattern is detected through forbidden dependency references (C_{forbid}) and explicit cycle analysis of the module dependency graph.
- *Overloaded Module APIs:* Uncontrolled growth of a module's public surface leads to unstable interfaces and defensive design. Refactoring becomes increasingly risky, and internal structure gradually freezes. This failure mode is indicated by a declining API-only dependency ratio (ρ_{api}) and sustained growth of the module's public API surface over time.
- *Feature-Centric Coupling Across Modules:* When features span multiple modules without clear ownership, features rather than modules become the primary unit of change. This pattern increases coordination cost and dilutes accountability. It is weakly visible in static dependencies but becomes evident through repeated cross-module co-changes and elevated architectural drift incidents.

These anti-patterns emerge when local convenience is prioritized over long-term change cost. G2 addresses them by enforcing explicit contracts, constraining dependency growth, and making structural erosion observable before it becomes irreversible.

Metrics and Verification

Maintainability in G2 is assessed through a set of explicit, measurable structural signals that capture boundary erosion, hidden coupling, and architectural drift over time. These metrics operationalize maintainability as a bounded cost of change by making the growth of coupling, dependency instability, and encapsulation violations observable and verifiable.

G2 builds directly on the enforcement model introduced in G1. While G1 establishes the correctness of modular boundaries, G2 uses quantitative signals to assess whether those boundaries remain effective as the system evolves. The quality measurement baseline derived from G1 is presented in Section **??** and is reused here as the foundation for maintainability verification.

- *Notation:* Let M be the set of bounded-context modules. Let R be the multiset of observed cross-module references, including static imports, dependency injection

wiring, and event handlers. Let $D \subseteq M \times M$ be the set of declared allowed dependencies (*requires*) and $F \subseteq M \times M$ be the set of explicitly forbidden dependencies (*forbids*).

- *Structural maintainability metrics:* The following metrics make boundary erosion and architectural drift observable using structural signals derived from cross-module references.
- *Undeclared Dependency Reference Count:*

$$C_{\text{undecl}} = |\{(A, B) \in R \mid (A, B) \notin D\}|$$

Maintainability meaning: measures hidden coupling introduced without architectural review.
Verification intent: ensure all cross-module dependencies are explicit and reviewable.

- *Forbidden Dependency Reference Count:*

$$C_{\text{forbid}} = |\{(A, B) \in R \mid (A, B) \in F\}|$$

Maintainability meaning: detects structurally disallowed coupling that violates architectural constraints.
Verification intent: preserve strict isolation where required and prevent dependency cycles.

- *API-Only Dependency Ratio:* Let $R_{\text{api}} \subseteq R$ be references targeting only the provider module's public surface:

$$\rho_{\text{api}} = \frac{|R_{\text{api}}|}{|R|}$$

Maintainability meaning: indicates whether dependencies are routed through stable contracts rather than internal structures.
Verification intent: constrain change propagation and stabilize interaction surfaces.

- *Encapsulation Leakage Count:* Let $R_{\text{internal}} \subseteq R$ be references to internal packages or non-exported symbols:

$$C_{\text{leak}} = |R_{\text{internal}}|$$

Maintainability meaning: quantifies boundary bypass that increases blast radius and refactoring risk.
Verification intent: enforce information hiding at the module boundary.

- *Module Isolation Test Pass Rate:* Let T_{iso} be the set of module isolation tests:

$$P_{\text{iso}} = \frac{|T_{\text{pass}}|}{|T_{\text{iso}}|}$$

Maintainability meaning: detects implicit coupling not visible through static analysis alone.
Verification intent: ensure modules can execute using only their declared dependencies.

- *Event Subscription Boundary Violations (when events are used):* Let H be the set of event handlers in module A consuming events from module B:

$$C_{\text{event}} = |\{(A, B) \in H \mid (A, B) \notin D\}|$$

Maintainability meaning: captures undeclared coupling introduced through event-driven integration.
Verification intent: keep asynchronous collaboration explicit and auditable.

- *Boundary Bypass Surface Count (optional):* Let S_{bypass} be occurrences of bypass mechanisms within cross-module interaction paths:

$$C_{\text{bypass}} = |S_{\text{bypass}}|$$

Maintainability meaning: indicates reliance on mechanisms that evade static boundary enforcement.
Verification intent: identify high-risk interaction paths that undermine architectural guarantees.

- *Supplementary maintainability metrics:* Some maintainability risks emerge over time and are not fully captured by static dependency analysis alone. G2 therefore defines the following supplementary metrics to detect API inflation and feature-centric coupling.

- *Public API Surface Growth Rate:*

$$G_{\text{api}}(m) = \frac{|API_m(t)| - |API_m(t - \Delta t)|}{\Delta t}$$

Maintainability meaning: sustained growth indicates increasing downstream obligations and reduced refactoring freedom.
Action: consolidate contracts, remove accidental exposure, or split responsibilities.

- *Change Coupling Index (optional):* Let $C(A, B)$ denote the number of commits

where modules A and B change together within a defined time window:

$$CCI = \frac{\sum_{A \neq B} C(A, B)}{\sum_{A} C(A)}$$

Maintainability meaning: high values indicate feature-centric coupling and blurred ownership boundaries.
Action: reassign responsibilities, introduce clearer module contracts, or refactor feature orchestration logic.

- *Verification strategy and maintainability interpretation:* In G2, these metrics are treated as longitudinal signals of architectural health rather than isolated quality indicators. Increases in C_{undecl}, C_{leak}, or C_{forbid} indicate growing hidden coupling and rising cost of change. A declining ρ_{api} signals unstable dependency surfaces, while reductions in P_{iso} reveal erosion of modular isolation.
  By tracking these metrics over time and enforcing thresholds where appropriate, G2 enables early detection of architectural drift and supports timely refactoring while corrective actions remain localized and economically viable. The primary objective is not to eliminate change, but to ensure that its cost remains bounded as the system evolves.

Literature Support Commentary

Although maintainability is a central topic in software engineering, much of the literature treats it either as an abstract principle or as a collection of localized code quality metrics. Recent empirical studies and systematic reviews emphasize that maintainability improvements in modular monoliths are primarily driven by boundary discipline and dependency management rather than deployment decomposition alone [GRAVANIS2021DONT,BERRY2024ISITWORT

Evolutionary architecture research further argues that architectural properties such as maintainability must be preserved through objective, executable constraints rather than informal guidance [FORDPARSONS2017]. G2 synthesizes these insights by framing maintainability as bounded cost of change and by providing concrete design mechanisms and measurable signals grounded in the boundary enforcement model established by G1.

4.6 G3: Design for Progressive Scalability

This section presents G3, which focuses on designing modular monoliths so that scalability is achieved progressively rather than through premature distribution. In this dissertation, *progressive scalability* is defined as the capacity of a modular monolith to accommodate increasing load, data volume, and organizational growth through a deliberate sequence of architectural interventions, each proportional to the observed need, without requiring a wholesale migration to a distributed architecture. G3 operationalizes this capacity by identifying module-level scale points, introducing asynchronous boundaries where load demands diverge, and abstracting infrastructure dependencies so that extraction remains a controlled option rather than an emergency.

G3 completes the Architectural Design Dimension (G1–G3). While G1 enforces modular boundaries and G2 preserves maintainability through bounded cost of change, G3 ensures that the resulting modular structure can absorb growth without collapsing into either a monolithic bottleneck or a premature microservices decomposition. Together, these three guidelines establish the structural foundation upon which the Operational Fit, Organizational Alignment, and Guideline Orientation dimensions build.

Intent and Rationale

The prevailing narrative in cloud-native literature frames scalability as an inherently distributed concern: when the system must scale, it must be decomposed into independently deployable services [ARYA2024BEYOND, BLINOWSKI2022MONOLITHIC]. This framing creates a false dichotomy. Teams are implicitly told to choose between a monolith that cannot scale and microservices that can, ignoring the intermediate architectural states that a well-modularized monolith can occupy.

In practice, most early-stage systems do not face uniform scaling pressure. Load concentrates in specific modules, specific use cases exhibit disproportionate growth, and specific data domains expand faster than others. A system designed for progressive scalability acknowledges this unevenness and provides mechanisms to address it surgically, scaling the modules that need it while leaving the rest undisturbed.

G3 therefore treats scalability as a *gradient* rather than a binary switch. The guideline proposes that a modular monolith can traverse a scalability spectrum through deliberate, reversible interventions:

1. *Vertical optimization within the monolith:* resource tuning, query optimization, caching, and concurrency improvements applied to specific modules without structural change.
2. *Asynchronous decoupling of hot paths:* introducing message queues or event streams between modules whose throughput requirements diverge, converting synchronous bottlenecks into buffered pipelines.

3. *Data isolation for divergent modules:* separating persistence for modules whose data access patterns, volume, or consistency requirements no longer fit a shared schema.
4. *Selective extraction of bounded contexts:* deploying individual modules as independent services only when the preceding interventions are insufficient and the operational cost is justified.

This spectrum makes the migration to microservices an *option*, not an obligation. Each step is proportional to observed need, reversible if conditions change, and grounded in the boundary enforcement (G1) and maintainability discipline (G2) already established. The key insight is that a system with enforced modular boundaries and stable contracts is *already extraction-ready*; what G3 adds is the deliberate preparation of infrastructure seams—established from project inception—that make extraction low-risk when evidence warrants it.

Conceptual Overview

Progressive scalability is embedded by designing modules so that growth is absorbed incrementally:

- Each module's resource consumption, throughput profile, and data growth trajectory are observable and attributable, enabling evidence-based scaling decisions.
- Communication between modules with divergent scaling needs can transition from synchronous to asynchronous without rewriting business logic.
- Persistence is designed so that schema ownership can be narrowed from shared to module-scoped without data migration crises.
- Infrastructure dependencies (databases, caches, queues, external APIs) are accessed through abstractions that decouple business logic from deployment topology.

Applicability Conditions and Scope

G3 applies to modular monolith systems that satisfy the following conditions:

- Module boundaries are enforced and dependencies are explicit, as established by G1.
- Maintainability discipline is in place, ensuring that contracts are stable and change locality is preserved, as established by G2.
- The system is expected to grow in load, data volume, or team size, but the timing and distribution of that growth are uncertain.

G3 does not prescribe specific cloud providers, container orchestrators, or scaling technologies. Its scope is limited to architectural preparation: the design decisions and structural properties that make scaling interventions feasible, proportional, and reversible. The guideline is agnostic to whether the system ultimately remains a monolith, becomes

a set of microservices, or stabilizes at an intermediate state.

Objectives

- *Identify module-level scale points:* Determine which modules are likely to experience disproportionate load, data growth, or throughput demands, and make these scale points visible in the architecture.
- *Enable asynchronous decoupling:* Design inter-module communication so that synchronous call paths can be replaced with asynchronous channels without modifying domain logic or violating module contracts.
- *Establish data isolation from inception:* Structure persistence so that each module owns a dedicated schema from the first deployment, ensuring that extraction never requires data migration.
- *Abstract infrastructure dependencies:* Ensure that modules interact with infrastructure (databases, caches, message brokers, external services) through ports or interfaces that can be re-bound to different implementations without changing business logic.
- *Preserve extraction optionality:* Maintain the ability to extract any bounded context as an independent service, without this extraction being a prerequisite for scaling.

Key Principles

- *Scale where it hurts, not where it might:* Scaling interventions should be driven by observed bottlenecks, not by speculative anticipation. Module-level metrics (see Metrics section) provide the evidence base. Premature optimization of modules that are not under pressure introduces unnecessary complexity and operational cost.
- *Asynchronous boundaries as scalability seams:* The boundary between two modules becomes a scalability seam when their throughput requirements diverge. Introducing an asynchronous channel (event bus, message queue) at this seam absorbs load spikes in the producer without propagating backpressure to the consumer, and vice versa. The key constraint is that asynchronous boundaries must respect the contracts established in G1 and the stability properties maintained by G2.
- *Schema isolation is established from project inception:* Each module owns a dedicated PostgreSQL schema from the first deployment. Cross-module data access is mediated exclusively through published contracts (events or API calls), never through direct table queries. This design decision, made at project inception rather than deferred to a future migration, ensures that data isolation is a structural property of the system rather than a prerequisite to be satisfied before extraction.
- *Infrastructure as a replaceable dependency:* Modules should depend on infrastructure through ports (interfaces) rather than concrete implementations. This is not an

abstract design preference but a scalability prerequisite: when a module must move from an in-process event bus to a distributed message broker, or from a shared PostgreSQL instance to a dedicated read replica, the change should be confined to the infrastructure adapter, not the domain logic.

- *Extraction is the last resort, not the first reflex:* Extracting a module into an independent service introduces network boundaries, distributed failure modes, deployment complexity, and operational overhead. G3 positions extraction at the end of the scalability spectrum, after vertical optimization, async decoupling, and data isolation have been exhausted or proven insufficient. This ordering preserves system simplicity for as long as possible while ensuring that extraction remains feasible when genuinely needed.

The Progressive Scalability Spectrum

To make the gradient concrete, G3 proposes a four-level spectrum that teams can use to assess their current position and plan their next proportional intervention. Each level builds on the previous one, and progression is driven by evidence rather than anticipation.

Table 2: Progressive Scalability Spectrum for Modular Monoliths

Level	Intervention	Description	Trigger Condition
L0	Vertical Optimization	Resource tuning, caching, query optimization, concurrency improvements within the monolith. No structural change.	Module-level latency or throughput degradation detected.
L1	Async Decoupling	Replace synchronous inter-module calls with asynchronous channels (event bus, message queue) for modules with divergent throughput.	Synchronous calls between modules create backpressure or latency spikes under load.
L2	Data Isolation	Separate module-scoped persistence (dedicated schemas, read replicas, or separate databases) for modules with divergent data access patterns.	Shared database contention, conflicting consistency requirements, or schema evolution friction.
L3	Selective Extraction	Deploy a bounded context as an independent service with its own runtime, persistence, and deployment pipeline.	L0–L2 exhausted; module requires independent scaling, independent release cadence, or technology heterogeneity.

The spectrum is not strictly linear: a team may apply L2 (data isolation) to one module while another module remains at L0. The key constraint is that each intervention at a

given level should be justified by evidence that the previous level is insufficient for the module in question.

Reference Implementation: Built-In Scalability Infrastructure

The Tiny Store reference implementation demonstrates that progressive scalability infrastructure is embedded from day one as an architectural requirement, not added later as a scaling patch. The following code listings show the actual production code that ships with the initial deployment.

L0: Vertical Optimization — Module-Scoped Caching

Tiny Store includes a Redis-backed caching layer from day one, not as an optimization added under pressure. The `CacheService` enforces module-namespaced keys, ensuring that cache entries are isolated per bounded context and can be invalidated independently.

Listing 16: CacheService with module-namespaced keys (cache.service.ts)

```
export class CacheService {
  private client: Redis;
  private defaultTtl: number;
  private globalPrefix: string;

  private buildKey(module: string, key: string): string {
    return `${this.globalPrefix}:${module}:${key}`;
  }

  async get<T>(module: string, key: string): Promise<T | null> {
    const raw = await this.client.get(this.buildKey(module, key));
    if (raw === null) return null;
    return JSON.parse(raw) as T;
  }

  async set<T>(module: string, key: string, value: T,
               ttlSeconds?: number): Promise<void> {
    const fullKey = this.buildKey(module, key);
    const ttl = ttlSeconds ?? this.defaultTtl;
    await this.client.set(fullKey, JSON.stringify(value), 'EX', ttl);
  }

  async invalidatePattern(module: string, pattern: string): Promise<void>
    {
    const keys = await this.client.keys(this.buildKey(module, pattern));
```

```
    if (keys.length > 0) await this.client.del(...keys);
  }
}
```

A read-through caching decorator applies this pattern declaratively to any module method:

Listing 17: Read-through caching decorator (cache.decorator.ts)

```
export function Cacheable(
  module: string, ttlSeconds: number,
  keyFn: (...args: any[]) => string,
) {
  return function (_target: any, _propertyKey: string,
                   descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = async function (...args: any[]) {
      const cache = CacheService.getInstance();
      const key = keyFn(...args);
      const cached = await cache.get(module, key);
      if (cached !== null) return cached;
      const result = await original.apply(this, args);
      await cache.set(module, key, result, ttlSeconds);
      return result;
    };
    return descriptor;
  };
}
// Usage: @Cacheable('inventory', 60, (sku) => `product:${sku}`)
```

The key design decision is that the cache layer exists from the first deployment. When vertical optimization becomes necessary (L0 trigger), the team enables caching on specific queries by adding a decorator—no infrastructure provisioning or architectural change is required.

L1: Async Decoupling — Module-Scoped Queues

Asynchronous processing is an architectural requirement, not a scaling patch. The `QueueService` provides module-scoped BullMQ queues following the convention `module:purpose`:

Listing 18: QueueService with module-scoped queues (queue.service.ts)

```
export class QueueService {
  private queues: Map<string, Queue> = new Map();
  private workers: Map<string, Worker> = new Map();
```

```
  /** Convention: `module:purpose` e.g. `orders:fulfillment` */
  createQueue(name: string): Queue {
    if (this.queues.has(name)) return this.queues.get(name)!;
    const queue = new Queue(name, { connection: getQueueConnection() });
    this.queues.set(name, queue);
    return queue;
  }

  async addJob<T>(queueName: string, data: T,
                  opts?: JobsOptions): Promise<Job<T>> {
    const queue = this.createQueue(queueName);
    return queue.add(queueName, data, opts);
  }

  createWorker<T>(queueName: string, handler: Processor<T>): Worker<T> {
    if (this.workers.has(queueName))
      return this.workers.get(queueName)! as Worker<T>;
    const worker = new Worker<T>(queueName, handler, {
      connection: getQueueConnection(),
    });
    this.workers.set(queueName, worker);
    return worker;
  }
}
```

Listing 19 shows a concrete use case: order confirmation emails are enqueued asynchronously with exponential backoff, decoupling the critical path (order placement) from the non-critical side effect (email delivery).

Listing 19: Order confirmation email job with async decoupling (order-confirmation-email.job.ts)

```
const QUEUE_NAME = 'orders:confirmation-email';

export function registerOrderConfirmationWorker(): void {
  const queueService = QueueService.getInstance();
  queueService.createWorker<OrderConfirmationData>(
    QUEUE_NAME,
    async (job: Job<OrderConfirmationData>) => {
      const { orderId, customerEmail, orderTotal } = job.data;
      // In production: await emailService.send(...)
      return { sent: true, orderId };
```

```
    },
  );
}

export async function enqueueOrderConfirmation(
  data: OrderConfirmationData,
): Promise<void> {
  const queueService = QueueService.getInstance();
  await queueService.addJob(QUEUE_NAME, data, {
    attempts: 3,
    backoff: { type: 'exponential', delay: 1000 },
  });
}
```

Because BullMQ and Redis are present from day one, the team can introduce async decoupling for any inter-module interaction by defining a new queue—no infrastructure migration is required.

L2: Data Isolation — Per-Module PostgreSQL Schemas

Schema isolation is a design decision made at project inception. Each module owns its data from the first deployment, ensuring that extraction never requires data migration. The `schema-isolation.ts` module creates per-module PostgreSQL schemas and provides module-scoped `DataSource` connections with isolated `search_path`:

Listing 20: Per-module PostgreSQL schema isolation (schema-isolation.ts)

```
const MODULE_SCHEMAS = ['orders', 'inventory', 'payments', 'shipments'];
export type ModuleName = (typeof MODULE_SCHEMAS)[number];

export async function createAllModuleSchemas(
  dataSource: DataSource,
): Promise<void> {
  for (const mod of MODULE_SCHEMAS) {
    await dataSource.query(`CREATE SCHEMA IF NOT EXISTS "${mod}"`);
  }
}

export async function getModuleConnection(
  baseOptions: DataSourceOptions,
  moduleName: ModuleName,
  entities: Function[],
): Promise<DataSource> {
```

```
  const options: DataSourceOptions = {
    ...baseOptions,
    name: `module-${moduleName}`,
    schema: moduleName,
    entities,
    ...(baseOptions.type === 'postgres' ? {
      extra: {
        ...baseOptions.extra,
        options: `-c search_path="${moduleName}",public`,
      },
    } : {}),
  } as DataSourceOptions;

  const ds = new DataSource(options);
  await ds.initialize();
  return ds;
}
```

Each module operates within its own PostgreSQL schema from the first `docker-compose up`. Cross-module data access is mediated through published contracts (events or API calls), never through direct table queries. When L3 extraction occurs, the module's schema migrates to a dedicated database instance—a deployment topology change, not a data migration.

Extraction Readiness Score

The readiness score provides the evidence that drives progression decisions. Tiny Store includes an automated extraction readiness calculator that scores each module across five dimensions:

Listing 21: Extraction readiness scoring dimensions (extraction-readiness.ts)

```
// Scoring dimensions (total: 100 points)
const checks: CheckResult[] = [
  checkCrossModuleImports(moduleName),  // 25pts: 0 imports = full score
  checkEventBusUsage(moduleName),       // 20pts: events + listeners
  checkACLLayer(moduleName),            // 15pts: anti-corruption layer
  checkDatabaseSchema(moduleName),      // 20pts: dedicated schema
  checkVersionedEvents(moduleName),     // 20pts: versioned contracts
];

// Readiness thresholds
if (percentage >= 80) console.log('READY for extraction');
```

```
else if (percentage >= 60) console.log('NEARLY READY');
else if (percentage >= 40) console.log('PARTIAL - significant work
    needed');
else console.log('NOT READY - major refactoring required');
```

Example output for the orders module:

```
Extraction Readiness Score: orders
==================================================
Cross-module imports [     ] 25/25
  No cross-module imports found
Event-driven communication [     ] 20/20
  Events: yes, Listeners: yes
ACL layer [     ] 15/15
  ACL adapter found for orders
Database schema isolation [     ] 20/20
  Schema 'orders' configured
Versioned event contracts [     ] 10/20
  Event contracts exist but no versioning
==================================================
TOTAL SCORE: 90/100 (90/100)
READY for extraction
```

The readiness score quantifies the ε_m metric defined in the Metrics section. Teams run this check before any L3 extraction decision, ensuring that extraction is evidence-based rather than intuition-driven.

Implementation Mechanisms

G3 is implemented through architectural preparation that is established from project inception, not added incrementally as scaling pressure appears. The following code listings from the Tiny Store reference implementation demonstrate the concrete mechanisms at each spectrum level.

L1: Module-Namespaced Caching

The `CacheService` provides a Redis-backed caching layer with module-namespaced keys, ensuring that cache entries are isolated per bounded context. The `buildKey` method constructs keys using the pattern `prefix:module:key`, preventing cross-module cache collisions and enabling per-module cache invalidation.

Listing 22: CacheService with module-namespaced keys (L1 vertical optimization)

```
// libs/shared/infrastructure/src/cache/cache.service.ts
export class CacheService {
  private client: Redis;
  private defaultTtl: number;
  private globalPrefix: string;

  private buildKey(module: string, key: string): string {
    return `${this.globalPrefix}:${module}:${key}`;
  }

  async get<T>(module: string, key: string): Promise<T | null> {
    const raw = await this.client.get(this.buildKey(module, key));
    if (raw === null) return null;
    return JSON.parse(raw) as T;
  }

  async set<T>(module: string, key: string, value: T,
               ttlSeconds?: number): Promise<void> {
    const fullKey = this.buildKey(module, key);
    const ttl = ttlSeconds ?? this.defaultTtl;
    await this.client.set(fullKey, JSON.stringify(value), 'EX', ttl);
  }

  async invalidatePattern(module: string, pattern: string): Promise<void>
    {
    const keys = await this.client.keys(this.buildKey(module, pattern));
    if (keys.length > 0) await this.client.del(...keys);
  }
}
```

Every cache call requires the module name as the first argument, making module attribution automatic. When the orders module caches a product lookup, the key becomes `tiny-store:orders:product:123`, fully isolated from inventory's `tiny-store:inventory:product:12`

L2 Async: Module-Scoped Queues with BullMQ

The `QueueService` creates module-scoped BullMQ queues following the convention `module:purpose`. This design establishes asynchronous decoupling as an architectural requirement from day one, not a migration step added when load increases.

Listing 23: QueueService with module-scoped queues (L2 async decoupling)

```
// libs/shared/infrastructure/src/queue/queue.service.ts
export class QueueService {
  private queues: Map<string, Queue> = new Map();
  private workers: Map<string, Worker> = new Map();

  /** Convention: `module:purpose` e.g. `orders:fulfillment` */
  createQueue(name: string): Queue {
    if (this.queues.has(name)) return this.queues.get(name)!;
    const queue = new Queue(name, { connection: getQueueConnection() });
    this.queues.set(name, queue);
    return queue;
  }

  async addJob<T>(queueName: string, data: T, opts?: JobsOptions):
    Promise<Job<T>> {
    const queue = this.createQueue(queueName);
    return queue.add(queueName, data, opts);
  }

  createWorker<T>(queueName: string, handler: Processor<T>): Worker<T> {
    if (this.workers.has(queueName)) return this.workers.get(queueName)!
    as Worker<T>;
    const worker = new Worker<T>(queueName, handler, {
      connection: getQueueConnection(),
    });
    this.workers.set(queueName, worker);
    return worker;
  }
}
```

The order confirmation email job demonstrates a concrete use of async decoupling: email sending is offloaded to a BullMQ worker, with exponential backoff retry built in from day one.

Listing 24: Order confirmation email job with async decoupling

```
// libs/modules/orders/src/jobs/order-confirmation-email.job.ts
const QUEUE_NAME = 'orders:confirmation-email';

export function registerOrderConfirmationWorker(): void {
  const queueService = QueueService.getInstance();
  queueService.createWorker<OrderConfirmationData>(
```

```
    QUEUE_NAME,
    async (job: Job<OrderConfirmationData>) => {
      const { orderId, customerEmail, orderTotal } = job.data;
      console.log(`[OrderConfirmationEmail] Sending for ${orderId} to
    ${customerEmail}`);
      return { sent: true, orderId };
    },
  );
}

export async function enqueueOrderConfirmation(
  data: OrderConfirmationData): Promise<void> {
  const queueService = QueueService.getInstance();
  await queueService.addJob(QUEUE_NAME, data, {
    attempts: 3,
    backoff: { type: 'exponential', delay: 1000 },
  });
}
```

L2 Data: Per-Module Schema Isolation

Schema isolation is built into the monolith from project inception. Each module receives its own PostgreSQL schema, and the `getModuleConnection` function returns a `DataSource` scoped to that schema via `search_path`. This makes data ownership ($\omega = 1$) an enforceable property rather than an aspiration.

Listing 25: Database schema isolation per module (L2 data isolation)

```
// libs/shared/infrastructure/src/database/schema-isolation.ts
const MODULE_SCHEMAS = ['orders', 'inventory', 'payments', 'shipments']
    as const;
export type ModuleName = (typeof MODULE_SCHEMAS)[number];

export async function createModuleSchema(
  dataSource: DataSource, moduleName: ModuleName): Promise<void> {
  await dataSource.query(`CREATE SCHEMA IF NOT EXISTS "${moduleName}"`);
}

export async function getModuleConnection(
  baseOptions: DataSourceOptions,
  moduleName: ModuleName,
  entities: Function[],
```

```
): Promise<DataSource> {
  const options: DataSourceOptions = {
    ...baseOptions,
    name: `module-${moduleName}`,
    schema: moduleName,
    entities,
    ...(baseOptions.type === 'postgres' ? {
      extra: {
        ...(baseOptions as any).extra,
        options: `-c search_path="${moduleName}",public`,
      },
    } : {}),
  } as DataSourceOptions;

  const ds = new DataSource(options);
  await ds.initialize();
  return ds;
}
```

Extraction Readiness Scoring

The extraction readiness tool computes a composite score (0–100) for any module by checking five dimensions that map directly to G1–G3 metrics. The tool is run from the command line and produces an actionable report.

Listing 26: Extraction readiness score calculator (composite metric)

```
// tools/metrics/extraction-readiness.ts
interface CheckResult {
  name: string;
  score: number;
  maxScore: number;
  details: string;
}

// Five check dimensions (25+20+15+20+20 = 100 max):
// 1. Cross-module imports      (25pts) - 0 violations = full score
// 2. Event-driven communication (20pts) - events + listeners
// 3. ACL layer                  (15pts) - anti-corruption adapter exists
// 4. Database schema isolation  (20pts) - dedicated schema configured
// 5. Versioned event contracts  (20pts) - version field in events
```

```
// Usage: npx ts-node tools/metrics/extraction-readiness.ts --module
    orders
// Output:
//   Cross-module imports  [@@@@@@@@@@] 25/25  No cross-module imports
//   Event-driven comm.    [@@@@@@@@@@] 20/20  Events: Y, Listeners: Y
//   ACL layer             [@@@@@@@@@@] 15/15  ACL adapter found
//   Database schema       [@@@@@@@@@@] 20/20  Schema 'orders' configured
//   Versioned events      [@@@@@@@@@@] 20/20  Versioned events found
//   TOTAL: 100/100 -- READY for extraction
```

G3 is implemented through the following architectural mechanisms, all established from project inception:

- *Module-scoped resource profiling:* Each module's resource consumption (CPU, memory, I/O, query volume) is attributable through instrumentation or naming conventions, enabling identification of scale points without distributed tracing.
- *Communication abstraction layer:* Inter-module communication passes through a contract-based abstraction (e.g., an event bus interface or a command dispatcher) that can be backed by in-process dispatch, an in-memory queue, or a distributed broker without changing the caller or handler.
- *Persistence port pattern:* Each module accesses its data through repository interfaces (ports) backed by a dedicated PostgreSQL schema from project inception. The concrete implementation is injected at composition time; when a module is extracted (L3), the repository adapter is re-bound to a dedicated database instance without changing domain logic. Schema ownership is documented in the module descriptor introduced in G1.
- *Scalability decision records:* When a module transitions between spectrum levels (e.g., from L0 to L1), the decision is recorded in an Architecture Decision Record (ADR) that captures the trigger condition, the intervention applied, the expected outcome, and the rollback path. This ensures that scaling decisions remain traceable and reversible.

Common Failure Modes and Anti-Patterns

- *Premature Distribution ("Microservices Envy"):* Extracting modules into services before exhausting in-process optimizations. This introduces distributed-system complexity (network latency, partial failures, eventual consistency) without evidence that distribution is necessary. The cost is disproportionate to the benefit, particularly for early-stage teams with limited operational maturity [GRAVANIS2021DONT, SU2024FROM].
- *Uniform Scaling Assumption:* Treating all modules as having identical scaling re-

quirements and applying the same intervention (e.g., horizontal replication of the entire monolith) uniformly. This wastes resources on modules that are not under pressure and delays targeted intervention for the modules that are.

- *Shared Database as Implicit Coupling:* Allowing modules to query each other's tables directly, creating an invisible dependency that prevents data isolation (L2) and makes extraction (L3) require a coordinated schema migration. This anti-pattern often accumulates silently because it does not violate code-level boundary checks.
- *Synchronous Call Chains Under Load:* Maintaining synchronous inter-module communication paths for workflows that exhibit high throughput variance. Under load, synchronous chains propagate latency and failures upstream, creating cascading degradation that appears as a system-wide outage rather than a localized bottleneck.
- *Infrastructure Lock-In Through Concrete Dependencies:* Embedding infrastructure-specific code (e.g., direct database driver calls, cloud SDK invocations) in domain or application logic. This prevents the module from transitioning between spectrum levels without rewriting business logic, effectively freezing the architecture at its current scale point.

Metrics and Verification

G3 metrics are designed to make scaling pressure observable at the module level and to verify that the architectural preparation for progressive scalability is in place. They complement the structural metrics defined in G1 and G2 by adding resource, throughput, and isolation dimensions.

- *Module Throughput Profile (Θ_m):* For each module m, the sustained request rate (operations per second) under normal and peak conditions. Divergence between modules indicates candidates for async decoupling (L1).

$$\Delta\Theta = \max_{m \in M} \Theta_m - \min_{m \in M} \Theta_m$$

  Scalability meaning: high $\Delta\Theta$ indicates uneven load distribution and potential synchronous bottlenecks.

- *Module Latency Variance ($\sigma^2_{L,m}$):* The variance of response latency for operations within module m. High variance under load suggests resource contention or synchronous dependency on a slower module. *Scalability meaning:* sustained increase in $\sigma^2_{L,m}$ is a trigger for L0 or L1 intervention.

- *Cross-Module Synchronous Call Ratio (ρ_{sync}):* The proportion of inter-module interactions that are synchronous (blocking) versus asynchronous (event-driven or queued):

$$\rho_{\text{sync}} = \frac{|I_{\text{sync}}|}{|I_{\text{sync}}| + |I_{\text{async}}|}$$

where I_{sync} and I_{async} are the sets of synchronous and asynchronous inter-module interactions, respectively. *Scalability meaning:* a high ρ_{sync} in the presence of divergent throughput profiles signals backpressure risk. Target: decrease ρ_{sync} for module pairs with high $\Delta\Theta$.

- *Data Ownership Clarity Index (ω):* The proportion of database tables (or collections) that are accessed by exactly one bounded-context module through its own repository:

$$\omega = \frac{|T_{\text{single-owner}}|}{|T|}$$

where T is the set of all tables and $T_{\text{single-owner}}$ is the subset accessed by only one module. *Scalability meaning:* $\omega < 1$ indicates shared tables that will block data isolation (L2) and extraction (L3). Target: $\omega \to 1$ over time.

- *Infrastructure Abstraction Coverage (α):* The proportion of infrastructure access points (database connections, cache clients, message broker producers/consumers, external API clients) that are mediated through an interface or port rather than used directly:

$$\alpha = \frac{|A_{\text{abstracted}}|}{|A|}$$

Scalability meaning: $\alpha < 1$ indicates infrastructure lock-in that will require code changes when transitioning between spectrum levels. Target: $\alpha = 1$ for modules identified as scale-point candidates.

- *Extraction Readiness Score (ε_m):* A composite indicator for module m that combines boundary enforcement (from G1), maintainability (from G2), and scalability preparation (from G3):

$$\varepsilon_m = f\Big(C^{(m)}_{\text{undecl}},\ C^{(m)}_{\text{leak}},\ \omega_m,\ \alpha_m,\ \rho^{(m)}_{\text{sync}}\Big)$$

where $C^{(m)}_{\text{undecl}}$ and $C^{(m)}_{\text{leak}}$ are the G1/G2 violation counts scoped to module m, and ω_m, α_m, $\rho^{(m)}_{\text{sync}}$ are the G3 metrics scoped to module m. *Scalability meaning:* a high ε_m indicates that the module can be extracted with low risk if L3 is triggered. A low ε_m signals preparation debt that must be addressed before extraction is viable.

Verification strategy: G3 metrics are tracked longitudinally alongside G1 and G2 metrics. The primary verification concern is not whether the system scales today, but whether it *can* scale proportionally when needed. An increase in $\Delta\Theta$ without a corresponding decrease in ρ_{sync} for the affected module pair indicates that scalability preparation is lagging behind actual load growth. A declining ω indicates increasing data coupling that will block future isolation. These signals enable proactive architectural intervention before scaling becomes an emergency.

Documentation Guidelines

- *Module Scale Profile:* Maintain a lightweight document (or section in the module descriptor) for each module that records its current spectrum level (L0–L3), its observed throughput profile, and any known scaling constraints. This profile is updated when monitoring data changes significantly or when a spectrum-level transition occurs.
- *Scalability ADRs:* Record each transition between spectrum levels as an Architecture Decision Record. The ADR should capture: the trigger condition (which metric crossed which threshold), the intervention applied, the expected outcome, the rollback path, and the actual outcome after implementation. This creates a traceable history of scaling decisions that can inform future interventions.
- *Infrastructure Dependency Map:* Document which infrastructure dependencies each module uses and whether they are accessed through abstractions ($\alpha = 1$) or directly ($\alpha < 1$). This map complements the module dependency descriptor from G1 and makes infrastructure lock-in visible.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to support progressive scalability should address:

- *Module-scoped metrics collection:* Attribute resource consumption, latency, and throughput to individual modules within the monolith, without requiring distributed tracing infrastructure.
- *Communication abstraction:* Provide a dispatch mechanism that supports both synchronous and asynchronous inter-module communication through the same contract interface.
- *Schema ownership analysis:* Identify tables or collections accessed by multiple modules and quantify data ownership clarity (ω).
- *Load simulation:* Support targeted load testing of individual modules or inter-module communication paths to validate scaling interventions before production deployment.

- *Extraction readiness assessment:* Combine G1 boundary metrics, G2 maintainability metrics, and G3 scalability metrics into a composite readiness view per module.

Literature Support Commentary

Scalability in the academic literature is predominantly framed as a property of distributed systems. Studies on microservices scalability [arya2024beyond, blinowski2022monolithic, berry2024isItWorth] provide empirical evidence on horizontal scaling, but consistently assume that distribution is a prerequisite. Conversely, studies that advocate for monoliths or modular monoliths [montesi2021sliceable, gravanis2021dont, de-Lauretis2019from] acknowledge that monoliths can serve moderate scaling needs but rarely formalize *how* scaling should be approached incrementally within a monolithic boundary.

Montesi et al. [montesi2021sliceable] come closest to the progressive scalability concept with their "sliceable monolith" proposal, which keeps the system unified until a slice needs independent deployment. However, their model focuses on deployment granularity rather than on the intermediate architectural interventions (async boundaries, data isolation) that G3 formalizes. Similarly, Ghemawat et al. [ghemawat2023towards] propose a "write as a monolith, deploy as microservices" model (Service Weaver), which addresses the deployment dimension but does not provide guidance on how to prepare the monolith's internal architecture for selective scaling.

The gap that G3 addresses is the absence of a structured, evidence-driven framework for scaling modular monoliths progressively. Current literature offers a binary choice: stay monolithic (and accept scaling limits) or migrate to microservices (and accept distributed complexity). G3 fills this void by defining intermediate interventions, trigger conditions for each intervention, and measurable indicators that make scalability preparation observable and verifiable within the existing modular structure. This approach aligns with the evolutionary architecture perspective [FordParsons2017] that architectural properties should be preserved through fitness functions and incremental adaptation rather than through disruptive transformations.

The scalability levels defined here provide the trigger criteria for G4 (Migration Readiness), which prepares individual modules for extraction.

Reference Implementation

The code listings in this section are drawn from the Tiny Store reference implementation. The relevant source files are:

- `libs/shared/infrastructure/src/cache/cache.service.ts` — Module-namespaced Redis caching (L0)

- `libs/shared/infrastructure/src/cache/cache.decorator.ts` — Read-through caching decorator (L0)
- `libs/shared/infrastructure/src/queue/queue.service.ts` — Module-scoped BullMQ queues (L1)
- `libs/modules/orders/src/jobs/order-confirmation-email.job.ts` — Async email decoupling (L1)
- `libs/shared/infrastructure/src/database/schema-isolation.ts` — Per-module PostgreSQL schemas (L2)
- `tools/metrics/extraction-readiness.ts` — Extraction readiness score calculator (ε_m)

Reference Implementation

All code listings in this section are drawn from the Tiny Store reference implementation. The relevant source files are:

- `libs/shared/infrastructure/src/cache/cache.service.ts` — CacheService with module-namespaced keys (Listing 22)
- `libs/shared/infrastructure/src/queue/queue.service.ts` — QueueService with module-scoped queues (Listing 23)
- `libs/modules/orders/src/jobs/order-confirmation-email.job.ts` — Async job example (Listing 24)
- `libs/shared/infrastructure/src/database/schema-isolation.ts` — Per-module schema isolation (Listing 25)
- `tools/metrics/extraction-readiness.ts` — Extraction readiness scoring (Listing 26)

Having established the scalability spectrum and progression criteria, the next guideline addresses how individual modules can be prepared for extraction when evidence warrants it.

4.7 G4: Promote Migration Readiness

This section presents G4, which focuses on designing modules so that they can be extracted from the monolith and deployed as independent services with minimal downstream disruption. Migration readiness is treated as a proactive design property rather than a reactive refactoring exercise. The guideline establishes the internal conditions, specifically internal APIs, anti-corruption layers, and the elimination of shared mutable state, that make extraction a controlled, low-risk operation when the conditions described in G3 (Level L3) are met.

G4 extends the Architectural Design Dimension (G1–G3) into the Operational Fit Dimension. While G1 enforces boundaries, G2 preserves maintainability, and G3 prepares for progressive scalability, G4 ensures that the transition from monolith to service, when justified, does not require a rewrite. The guideline targets the structural prerequisites that make the difference between a planned extraction and an emergency migration.

Intent and Rationale

Migration from a monolith to microservices is one of the most expensive architectural transitions a team can undertake. Studies consistently report that migration efforts are dominated not by the extraction itself, but by the discovery and resolution of hidden coupling: shared database tables, implicit assumptions about co-location, synchronous call chains that mask network-boundary failures, and domain logic scattered across modules [FRITZSCH2019, ABGAZ2023DECOMPOSITION, WOLFART2021MODERNIZING].

The root cause is that most monoliths are not designed with extraction in mind. Dependencies accumulate implicitly, persistence is shared by convention, and inter-module communication relies on in-process guarantees (transactions, shared memory, synchronous method calls) that do not survive a network boundary. When migration pressure arrives, typically driven by scaling needs (G3, Level L3), independent release cadence, or technology heterogeneity, teams discover that the actual cost of extraction is proportional to the amount of implicit coupling that must be made explicit.

G4 inverts this dynamic. Instead of treating migration as a future problem to be solved when it arises, G4 embeds migration readiness into the module's design from the outset. The cost is incremental: designing internal APIs, introducing anti-corruption layers at integration points, and avoiding shared mutable state are practices that improve the system's quality independent of whether extraction ever occurs. The benefit is optionality: when extraction is justified, the module is already prepared.

Conceptual Overview

Migration readiness is embedded by designing each module so that its external integration surface is network-boundary compatible. Crucially, G4 does not treat this as a hypothetical design exercise. The guideline prescribes introducing production-grade infrastructure, specifically Apache Kafka for event streaming and Temporal for durable workflow orchestration, *inside* the monolith, before any extraction occurs. This approach eliminates the gap between "designed for extraction" and "ready for extraction": the same brokers, workflow engines, and contract mechanisms that would be used in a distributed deployment are already running in the monolith, validated by the same integration tests, and observable through the same monitoring tools.

- Each module exposes its capabilities through a versioned internal API that could be replaced by a network endpoint without changing the contract.
- Integration points between modules are mediated by anti-corruption layers that translate between domain models, preventing one module's internal evolution from breaking another.
- No module relies on shared mutable state (shared database tables, in-memory caches, global singletons) for correctness; each module owns its data and collaborates through contracts.
- Cross-module workflows are orchestrated through Temporal, which provides durable execution, explicit compensation paths, and activity-level retry policies that function identically whether activities are in-process calls or network calls.
- Domain events are published to Kafka topics, providing durable, partitioned delivery with consumer group semantics that support horizontal scaling of listeners, both before and after extraction.

Applicability Conditions and Scope

G4 applies to modular monolith systems where:

- Module boundaries are enforced (G1) and maintainability discipline is in place (G2).
- At least some modules are expected to face extraction pressure in the medium term, due to scaling needs (G3), independent release cadence, or technology heterogeneity.
- The team prefers to invest incrementally in extraction readiness rather than accept a high-cost migration event later.

G4 does not prescribe when to extract. That decision belongs to G3's scalability spectrum (Level L3). G4 prescribes *how to be ready* when the decision is made. The guideline is technology-agnostic: the principles apply whether the system uses REST, gRPC, message queues, or any other communication mechanism.

Objectives

- *Design internal APIs as if they were external:* Module public surfaces should be defined with the same discipline applied to external APIs: versioned, documented, backward-compatible, and independent of internal implementation details.
- *Introduce anti-corruption layers at integration points:* Ensure that each module translates inbound and outbound data through an explicit mapping layer, so that internal model changes do not propagate across boundaries.
- *Eliminate shared mutable state:* Ensure no module depends on another module's database tables, in-memory state, or global resources for correctness.
- *Replace distributed transactions with sagas:* Design cross-module workflows as compensable sequences rather than atomic transactions, so that the same logic functions correctly across process boundaries.
- *Make extraction a configuration change, not a rewrite:* The ideal extraction involves changing infrastructure wiring (swapping in-process dispatch for network calls, pointing to a separate database) without modifying domain or application logic.

Key Principles

- *Internal APIs deserve the same rigor as external APIs:* In a modular monolith, the temptation is to treat inter-module interfaces as informal. Since everything runs in-process, breaking changes are discovered immediately and can be fixed in the same commit. This convenience is precisely what creates migration debt. When an internal API is designed with versioning, backward compatibility, and explicit contracts, extraction requires only a transport change (in-process call $\rightarrow$ HTTP/gRPC call), not an interface redesign.
- *Anti-corruption layers prevent model leakage:* Each module has its own domain model, its own vocabulary, and its own invariants. When module A consumes data from module B, it should not depend on B's internal representation. An anti-corruption layer (ACL) sits at the boundary and translates B's published model into A's internal model. This decouples the evolution of both modules: B can refactor its internals without breaking A, and A can evolve its model without requiring changes in B [EVANS2003DDD].
- *Data ownership is non-negotiable:* Shared database tables are the single largest obstacle to extraction. When two modules read from or write to the same table, they are coupled at the persistence level regardless of how clean their code-level boundaries are. G4 requires that each module owns its data exclusively. Cross-module data access occurs through the module's public API or through published events, never through direct database queries. This principle builds on G3's Data

Ownership Clarity Index (ω) and treats $\omega = 1$ as a prerequisite for extraction readiness.

- *Eventual consistency is the default for cross-module workflows:* Monoliths often rely on database transactions to maintain consistency across modules. This works in-process but fails across network boundaries. G4 replaces cross-module transactions with saga patterns: each module performs its local operation and publishes an event; downstream modules react and either continue the workflow or trigger a compensating action. The saga pattern is already implicit in Tiny Store's event-driven choreography (OrderPlaced → InventoryReserved → PaymentProcessed → ShipmentCreated); G4 makes it explicit and robust.
- *Infrastructure wiring is the extraction knob:* When a module is ready for extraction, the change should be localized to the composition root: swap the in-process event bus for a distributed broker, replace the shared database connection with a dedicated one, and route API calls through a network client instead of a direct import. If extraction requires changes to domain logic, the module is not yet migration-ready.

The Orders Module: A Production-Grade Migration Readiness Case Study

The previous guidelines (G1–G3) used Tiny Store as a teaching artifact, a small modular monolith designed to make architectural concepts observable and testable. G4 shifts the register. This case study takes the orders module beyond pedagogical illustration and prepares it with the same infrastructure a production e-commerce system would use: Apache Kafka for durable event streaming, Temporal for workflow orchestration and saga management, schema-level data isolation, and versioned event contracts with anti-corruption layers. The intent is to demonstrate that migration readiness is not a theoretical exercise, but a concrete engineering practice that produces a system indistinguishable from a production-ready architecture, even while it remains a monolith.

In any e-commerce system, order placement is the highest-traffic path: every checkout invokes it, and it fans out to inventory reservation, payment processing, and shipment creation. As the business scales, orders is likely to be the first module to hit the scaling limits described in G3, making it the most probable extraction candidate. The case study examines the orders module through the lens of G4's five objectives, identifying what is already migration-ready and what would need to change to reach production-grade extraction readiness.

Current State: What Orders Already Does Well

The orders module in Tiny Store already satisfies several migration readiness properties by construction:

- *Explicit public surface:* The module exposes capabilities through `@tiny-store/modules-orders`, which re-exports only handlers and event contracts. Internal entities (`Order`, `OrderRepository`) are not exported.
- *Event-driven collaboration:* Cross-module interaction follows the event choreography pattern. Orders publishes `OrderPlaced`, `OrderCancelled`, and other domain events; downstream modules react through listeners wired in the composition root. There are no direct imports between orders and inventory, payments, or shipments.
- *Centralized wiring:* All event subscriptions are registered in `register-listeners.ts`, making the integration surface auditable and modifiable from a single location.

These properties mean that, at the code level, extracting orders would not require rewriting cross-module interactions. The event contracts serve as the interface, and the composition root serves as the wiring point.

Migration Readiness Gaps: From Teaching Artifact to Production System

Despite strong boundary enforcement, Tiny Store's baseline state reflects a common pattern in early-stage monoliths: boundaries are clean at the code level, but the infrastructure layer still assumes co-location. The gaps below are not unique to Tiny Store. They appear in virtually every modular monolith that has not been deliberately prepared for extraction, and closing them is what transforms an architectural prototype into a production-ready system:

1. *Shared database:* All modules share a single database instance. The `orders` table and the `inventory` table live in the same schema. Even though code-level boundaries are clean, a direct SQL join or a shared transaction could couple them at the persistence level. Extraction requires that orders owns its own database (or at minimum, its own schema), and that any data it needs from inventory arrives through events or API calls.
2. *In-process event bus:* In a naive modular monolith, the event bus would be an in-memory singleton with synchronous delivery. Tiny Store avoids this gap by including Kafka from inception: the `EventBusFactory` (Listing 27) selects between in-memory and Kafka adapters via an environment variable. Each module publishes to a dedicated topic (e.g., `orders.events`), and downstream modules consume through consumer groups that support horizontal scaling and exactly-once processing semantics. This is a design requirement, not a migration step.
3. *Transactional assumptions:* When an order is placed, the system could benefit from the fact that inventory reservation happens in the same process. After extraction, the "place order → reserve inventory" flow must tolerate partial failures. G4 addresses this by including Temporal[5] as a durable workflow orchestrator from in-

[5] `https://temporal.io`

ception. Temporal persists workflow state automatically, provides built-in retry and timeout policies per activity, and executes compensating actions reliably even across process restarts. The workflow runs inside the monolith from day one, then survives extraction unchanged—this is an architectural requirement, not a migration step.

4. *Anti-corruption layers:* Tiny Store includes ACL gateways from inception (Listings 33 and 34). The orders module communicates with inventory and payments exclusively through gateway interfaces (`InventoryGateway`, `PaymentGateway`). The adapter implementations translate between domain models, so that swapping an in-process call for an HTTP client after extraction requires changing only the adapter, not the domain logic.
5. *No API versioning:* The orders module's public handlers do not carry version metadata. In a co-located monolith, this is acceptable because all consumers are updated simultaneously. After extraction, the orders service must support multiple API versions to allow independent deployment of consumers.

Preparing Orders for Extraction: Production-Grade Steps

The following steps describe how to bring the orders module from its baseline state to production-grade migration readiness, without extracting it. Unlike the G1 tutorial, which focused on boundary verification through lightweight test targets, these steps introduce production infrastructure: Kafka for event streaming, Temporal for durable workflow orchestration, schema isolation for data ownership, and versioned contracts for independent evolution. Each step is designed so that the resulting system is not merely "ready for extraction in theory" but operationally indistinguishable from a distributed setup, running the same brokers, the same workflow engine, and the same contract mechanisms that would be used after extraction. The only difference is deployment topology: everything still runs as a single deployable unit.

- *Step 1: Isolate the orders schema.* Create a dedicated database schema (or namespace) for orders. Migrate the `orders` and `order_items` tables into this schema. Update the orders repository to use the isolated schema. Verify that no other module queries these tables directly (G3's ω metric should increase toward 1).
- *Step 2: Abstract the event bus behind a port, with Kafka as the production adapter.* Kafka is included from inception as part of the modular monolith's infrastructure. The system defines an `IEventPublisher` interface (port) in the shared infrastructure layer, with two adapters: an in-memory `EventBus` for development and testing, and a `KafkaEventBusAdapter` for production. A factory function selects the adapter based on the `EVENT_TRANSPORT` environment variable. This makes event transport a configuration decision from day one, not a migration step.

Listing 27: Event bus factory with Kafka adapter (`event-bus.factory.ts`)

```
// libs/shared/infrastructure/src/event-bus/event-bus.factory.ts
export interface IEventPublisher {
  publish(event: DomainEvent): Promise<void>;
  subscribe(eventType: string,
    handler: (event: DomainEvent) => Promise<void> | void): void;
}

class KafkaEventBusAdapter implements IEventPublisher {
  private kafkaProducer: KafkaProducer;
  private localBus: EventBus;
  constructor() {
    this.kafkaProducer = KafkaProducer.getInstance();
    this.localBus = EventBus.getInstance();
  }
  async publish(event: DomainEvent): Promise<void> {
    await this.kafkaProducer.publish(event); // Kafka for
   cross-service
    await this.localBus.publish(event);      // local for in-process
  }
  subscribe(eventType: string,
    handler: (event: DomainEvent) => Promise<void> | void): void {
    this.localBus.subscribe(eventType, handler);
  }
}

export function createEventPublisher(): IEventPublisher {
  const transport = process.env.EVENT_TRANSPORT || 'memory';
  if (transport === 'kafka') return new KafkaEventBusAdapter();
  return EventBus.getInstance();
}
```

The Kafka producer publishes events to per-module topics (e.g., `orders.events`), with trace context propagation for distributed observability:

Listing 28: Kafka producer with trace propagation (`kafka.producer.ts`)

```
// libs/shared/infrastructure/src/kafka/kafka.producer.ts
export class KafkaProducer {
  private static instance: KafkaProducer;
  private producer: Producer;
  private connected = false;
```

```
  async publish(event: DomainEvent): Promise<void> {
    await this.ensureConnected();
    const topic = `${event.aggregateType}.events`;
    await this.producer.send({
      topic,
      messages: [{
        key: event.aggregateId,
        value: JSON.stringify({
          eventId: event.eventId, eventType: event.eventType,
          aggregateId: event.aggregateId,
          aggregateType: event.aggregateType,
          occurredAt: event.occurredAt.toISOString(),
          payload: event.payload, version: event.version,
        }),
        headers: injectTraceContext({
          eventId: event.eventId, eventType: event.eventType,
          correlationId: event.aggregateId,
        }),
      }],
    });
  }
}
```

On the consumer side, each downstream module subscribes to the relevant Kafka topic using a dedicated consumer group. The consumer includes idempotency checks and OpenTelemetry trace extraction:

Listing 29: Kafka consumer with idempotency and tracing (`kafka.consumer.ts`)

```
// libs/shared/infrastructure/src/kafka/kafka.consumer.ts
export class KafkaConsumer {
  private handlers: Map<string, Set<EventHandler>> = new Map();
  private processedEventIds: Set<string> = new Set();

  constructor(groupId: string) { /* ... */ }

  subscribe(eventType: string, handler: EventHandler): void {
    if (!this.handlers.has(eventType))
      this.handlers.set(eventType, new Set());
    this.handlers.get(eventType)!.add(handler);
  }

  async start(topics: string[]): Promise<void> {
```

```
    await this.consumer.connect();
    for (const topic of topics)
      await this.consumer.subscribe({ topic, fromBeginning: false });
    await this.consumer.run({
      eachMessage: async (msg) => this.handleMessage(msg),
    });
  }

  private async handleMessage({ message, topic }:
   EachMessagePayload) {
    const event = JSON.parse(message.value!.toString());
    if (this.processedEventIds.has(event.eventId)) return; //
   idempotent
    this.processedEventIds.add(event.eventId);
    const handlers = this.handlers.get(event.eventType);
    if (handlers)
      await Promise.all([...handlers].map((h) => h(event)));
  }
}
```

Topic names and consumer groups are centralized in a configuration file that ensures consistency across all modules:

Listing 30: Centralized topic configuration (`topics.config.ts`)

```
// libs/shared/infrastructure/src/kafka/topics.config.ts
export const TOPICS = {
  ORDERS:    'orders.events',
  INVENTORY: 'inventory.events',
  PAYMENTS:  'payments.events',
  SHIPMENTS: 'shipments.events',
} as const;

export const CONSUMER_GROUPS = {
  ORDERS:    'orders-consumer',
  INVENTORY: 'inventory-consumer',
  PAYMENTS:  'payments-consumer',
  SHIPMENTS: 'shipments-consumer',
} as const;
```

- *Step 3: Orchestrate the order lifecycle as a Temporal workflow.* Tiny Store's current event choreography (OrderPlaced → InventoryReserved → PaymentProcessed → ShipmentCreated) distributes saga logic implicitly across listeners. This works

in-process but becomes fragile after extraction: failure visibility is poor, retry policies are ad hoc, and compensating actions are scattered across modules. G4 replaces implicit choreography with an explicit orchestration using Temporal[6], a durable execution engine that provides: built-in retry and timeout policies per activity, automatic state persistence (the workflow survives process restarts), native compensation through saga patterns, and full observability of workflow execution history.

The key insight is that Temporal can be introduced *inside* the monolith first. The workflow orchestrator runs as a Temporal worker within the same process, invoking module handlers as Temporal activities. After extraction, the same workflow code runs unchanged; only the activity implementations switch from in-process calls to network calls (gRPC or HTTP). This makes Temporal both a migration readiness tool and a production-grade saga manager.

Listing 31: Order fulfillment saga workflow (`order-fulfillment.workflow.ts`)

```
// libs/modules/orders/src/workflows/order-fulfillment.workflow.ts
import { proxyActivities, ApplicationFailure } from
    '@temporalio/workflow';
import type * as activities from './order-fulfillment.activities';

const { reserveInventory, releaseInventory, processPayment,
  refundPayment, createShipment,
} = proxyActivities<typeof activities>({
  startToCloseTimeout: '30s',
  retry: { maximumAttempts: 3 },
});

export interface OrderFulfillmentInput {
  orderId: string;
  items: Array<{ sku: string; quantity: number; unitPrice: number }>;
  totalAmount: number;
  shippingAddress: { street: string; city: string; state: string;
    postalCode: string; country: string };
}

export async function orderFulfillmentWorkflow(
  input: OrderFulfillmentInput
): Promise<{ success: boolean; trackingNumber?: string; error?:
    string }> {
  // Step 1: Reserve inventory
  const reservation = await reserveInventory({
```

[6] https://temporal.io

```
    orderId: input.orderId,
    items: input.items.map((i) => ({ sku: i.sku, quantity:
   i.quantity })),
  });
  if (!reservation.success)
    return { success: false, error: reservation.error };

  // Step 2: Process payment (compensate on failure)
  let paymentResult;
  try {
    paymentResult = await processPayment({
      orderId: input.orderId, amount: input.totalAmount });
  } catch (err) {
    await releaseInventory({ orderId: input.orderId });
    throw ApplicationFailure.nonRetryable(
      `Payment failed for ${input.orderId}, inventory released`);
  }
  if (!paymentResult.success) {
    await releaseInventory({ orderId: input.orderId });
    return { success: false, error: paymentResult.errorMessage };
  }

  // Step 3: Create shipment (compensate on failure)
  let shipment;
  try {
    shipment = await createShipment({
      orderId: input.orderId,
      shippingAddress: input.shippingAddress });
  } catch (err) {
    await refundPayment({ orderId: input.orderId,
      paymentId: paymentResult.paymentId });
    await releaseInventory({ orderId: input.orderId });
    throw ApplicationFailure.nonRetryable(
      `Shipment failed for ${input.orderId}, compensated`);
  }
  return { success: true, trackingNumber: shipment.trackingNumber };
}
```

Listing 32: Temporal activities: in-process today, network calls after extraction (`order-fulfillment.activities.ts`)

```
// libs/modules/orders/src/workflows/order-fulfillment.activities.ts
```

```
// Activities call module APIs through ACL gateways.
// Today: in-process. After extraction: swap for HTTP/gRPC clients.
// The workflow code does NOT change.

export async function reserveInventory(
  input: ReserveInventoryInput
): Promise<ReserveInventoryResult> {
  // Calls inventory module via InventoryGateway ACL
  return { success: true, orderId: input.orderId };
}

export async function releaseInventory(
  input: { orderId: string }
): Promise<void> { /* compensating action */ }

export async function processPayment(
  input: { orderId: string; amount: number }
): Promise<ProcessPaymentResult> {
  // Calls payments module via PaymentGateway ACL
  return { paymentId: `pay-${input.orderId}`, success: true };
}

export async function refundPayment(
  input: { orderId: string; paymentId: string }
): Promise<void> { /* compensating action */ }

export async function createShipment(
  input: CreateShipmentInput
): Promise<CreateShipmentResult> {
  return { shipmentId: `ship-${input.orderId}`,
    trackingNumber: `TRK-${Date.now()}` };
}
```

The Temporal workflow provides several properties that pure event choreography lacks: (i) the full compensation path is visible in a single file rather than scattered across listeners; (ii) Temporal's execution history records every activity attempt, retry, and compensation, providing audit-grade observability; (iii) timeout and retry policies are declared per activity rather than implemented ad hoc in each listener; and (iv) the workflow is testable using Temporal's time-skipping test framework, which can simulate failures at any step and verify the correct compensation sequence.

- *Step 4: Introduce anti-corruption layers in downstream modules.* In the inventory module, create a translation layer that maps the `OrderPlaced` event payload into inventory's own internal command. This way, if orders changes its event schema (e.g., renames a field or restructures the payload), the inventory module's ACL absorbs the change without affecting inventory's domain logic.

Listing 33: ACL: Orders → Inventory gateway (`inventory-gateway.acl.ts`)

```
// libs/modules/orders/src/acl/inventory-gateway.acl.ts
export interface InventoryGateway {
  reserveItems(orderId: string, items: OrderItem[]):
    Promise<ReserveItemsResult>;
  releaseItems(orderId: string): Promise<void>;
}

// Default: calls inventory module in-process.
// After extraction: replace with HTTP/gRPC client.
export class InventoryGatewayAdapter implements InventoryGateway {
  constructor(deps: {
    reserveStock: (dto: {...}) => Promise<ReserveItemsResult>;
    releaseStock: (dto: {...}) => Promise<{...}>;
  }) { /* wire dependencies */ }

  async reserveItems(orderId: string, items: OrderItem[]) {
    return this.reserveStockFn({ orderId, items });
  }
  async releaseItems(orderId: string) {
    await this.releaseStockFn({ orderId });
  }
}
```

Listing 34: ACL: Orders → Payments gateway (`payment-gateway.acl.ts`)

```
// libs/modules/orders/src/acl/payment-gateway.acl.ts
export interface PaymentGateway {
  requestPayment(orderId: string, amount: number):
    Promise<PaymentResult>;
  requestRefund(orderId: string, paymentId: string): Promise<void>;
}

// Default: calls payments module in-process.
// After extraction: replace with HTTP/gRPC client.
export class PaymentGatewayAdapter implements PaymentGateway {
```

```
  constructor(deps: {
    processPayment: (dto: {...}) => Promise<{...}>;
  }) { /* wire dependencies */ }

  async requestPayment(orderId: string, amount: number) {
    const result = await this.processPaymentFn({ orderId, amount });
    return { paymentId: result.paymentId,
      success: result.success, errorMessage: result.errorMessage };
  }
  async requestRefund(orderId: string, paymentId: string) { /* ...
    */ }
}
```

- *Step 5: Add version metadata to the orders public API.* Annotate event contracts and handler interfaces with a version identifier. This does not require a full API gateway; it can be as simple as a version field in the event payload or a version constant exported alongside each handler. The goal is to enable backward-compatible evolution after extraction.

Listing 35: Versioned event contract with factory (`order-placed.event.ts`)

```
// libs/shared/contracts/src/events/order-placed.event.ts
import { DomainEvent } from '@tiny-store/shared-domain';
import { v4 as uuid } from 'uuid';

export interface OrderPlacedPayload {
  orderId: string;
  items: Array<{ sku: string; quantity: number; unitPrice: number }>;
  total: number;
  version: 1;  // explicit schema version for backward compatibility
}

export interface OrderPlacedEvent extends DomainEvent {
  eventType: 'OrderPlaced';
  payload: OrderPlacedPayload;
}

export function createOrderPlacedEvent(
  orderId: string,
  payload: Omit<OrderPlacedPayload, 'version'>
): OrderPlacedEvent {
  return {
```

```
    eventId: uuid(),
    eventType: 'OrderPlaced',
    aggregateId: orderId,
    aggregateType: 'Order',
    occurredAt: new Date(),
    version: 1,
    payload: { ...payload, version: 1 },
  };
}
```

Extraction Readiness Checklist for Orders

After completing Steps 1–5, the orders module operates on production-grade infrastructure (Kafka, Temporal, isolated schema) while remaining inside the monolith. The following checklist confirms that extraction, when triggered by G3's scalability spectrum (Level L3), requires only a deployment topology change:

	Prerequisite	**Status**
✓	Own database schema (no shared tables)	Step 1
✓	Event publishing through Kafka-backed port	Step 2
✓	Temporal workflow with compensating actions	Step 3
✓	ACLs in all downstream consumers	Step 4
✓	Versioned event contracts and public API	Step 5
✓	No direct cross-module imports (G1)	Baseline
✓	Centralized wiring in composition root	Baseline
✓	Module isolation tests pass (G1/G2)	Baseline

When all prerequisites are met, extracting orders means: deploying it as a standalone service with its own database, switching the `InMemoryEventPublisher` to the `KafkaEventPublisher` (pointing to the `orders.events` topic), replacing Temporal activity implementations from in-process handler calls to HTTP/gRPC client calls, and routing incoming requests through a network endpoint. The Temporal workflow code remains identical. The Kafka topic structure remains identical. The ACLs in downstream consumers remain identical. No domain logic changes. No saga redesign. No downstream module rewrites.

Operationalizing Extraction: The Extract Command

While the previous sections establish the theoretical foundations for migration readiness, production systems require practical tooling to operationalize these principles. Drawing inspiration from Service Weaver's configuration-driven deployment model [GHEMAWAT2023WEAVER], which treats distribution as a configuration concern rather than a coding concern, this

section presents the `extract` command—a tooling solution that automates the transition from monolith to service while preserving the infrastructure abstractions described in Steps 1–5.

The extract command embodies G4's core principle that extraction should be a configuration change, not a rewrite. Rather than requiring manual coordination across database migration, service scaffolding, Docker configuration, monolith updates, and deployment orchestration, the extract command executes these steps atomically through a declarative extraction manifest. The command is designed for the modular monoliths that have achieved migration readiness: modules with clean boundaries (G1), infrastructure abstraction through ports (G2/G3), and event-driven integration patterns that survive network boundaries.

The Extraction Manifest

The extract command operates from a declarative configuration file (`extraction.config.yml`) that specifies which modules are deployed as monoliths versus independent services. This manifest serves as the single source of truth for deployment topology, following Service Weaver's philosophy that the same application code should be deployable in different configurations without modification.

Listing 36: Extraction configuration manifest

```
version: '1.0'

modules:
  orders:
    mode: extracted              # monolith | extracted
    database:
      type: postgres
      name: tinystore_orders
      port: 5432
    service:
      port: 3001
      dockerfile: apps/orders-service/Dockerfile
    dependencies: [inventory, payments, shipments]

  inventory:
    mode: monolith

  payments:
    mode: monolith
```

```
  shipments:
    mode: monolith

infrastructure:
  kafka:
    brokers: ['localhost:9092']
    topics:
      orders: orders.events
      inventory: inventory.events
      payments: payments.events
      shipments: shipments.events

  temporal:
    address: localhost:7233
    namespace: tiny-store

  postgres:
    host: localhost
    port: 5432
    username: tiny
    password: store
```

The manifest captures the deployment intentions without embedding them in application code. A module's `mode` can transition from `monolith` to `extracted` through a configuration change, triggering the automated migration process. The infrastructure section specifies the Kafka brokers, Temporal namespace, and PostgreSQL connection parameters that support both deployment modes.

Automated Extraction Flow

The extract command orchestrates a multi-step process that transforms a module from in-process deployment to independent service deployment. Each step is designed to be atomic and reversible, ensuring that partial failures can be rolled back cleanly.

Listing 37: Extract command usage

```
# Extract the orders module as an independent service
npx ts-node tools/extract/extract.ts orders

# Show current extraction status
npx ts-node tools/extract/extract.ts status

# Retract the orders service back to monolith deployment
```

```
npx ts-node tools/extract/extract.ts retract orders
```

The extraction process consists of eight automated steps:

1. *Entity Discovery:* Parse TypeScript source files to identify `@Entity` decorators and their associated database tables. This step ensures that all database artifacts belonging to the module are discovered programmatically rather than maintained in a separate manifest.
2. *Database Migration:* Create a dedicated PostgreSQL database for the extracted service and migrate data from the shared SQLite database. This step materializes the data ownership principle (G3's $\omega = 1$) by providing the module with an isolated persistence layer.
3. *Service Scaffolding:* Generate a complete Express-based service application with TypeORM configuration, health check endpoints, and integration with the existing module handlers. The scaffolded service uses the same Temporal workflow definitions and Kafka event publishers as the monolith, ensuring behavioral equivalence.
4. *Monolith Updates:* Remove the extracted module's entities from the monolith's database configuration and install HTTP proxy routes that forward requests to the extracted service. This step ensures that existing consumers of the module's API continue to function without modification.
5. *Docker Configuration:* Generate Dockerfile, .dockerignore, and docker-compose service definitions for the extracted service, enabling containerized deployment with production-grade resource limits and health checks.
6. *Nx Workspace Integration:* Update the monorepo's TypeScript path mappings, package.json scripts, and project.json configurations to support building, testing, and serving the extracted service alongside the monolith.
7. *Infrastructure Wiring:* Update the extracted service's configuration to use Kafka producers instead of the in-memory event bus, and point Temporal activities to network endpoints instead of in-process handler calls.
8. *Verification:* Execute integration tests against both the monolith (with proxy routes) and the extracted service (with network boundaries) to verify that the extraction preserves functional correctness.

The Retract Command: Rollback as a First-Class Operation

Production systems require the ability to reverse architectural decisions when extraction proves premature or problematic. The `retract` command provides an automated rollback mechanism that restores the module to monolith deployment while preserving data and configuration history.

Listing 38: Retract command restoring monolith deployment

```
# Retract orders service back to monolith
npx ts-node tools/extract/extract.ts retract orders

# Output:
# [retry] Starting retraction of orders service...
# [metrics] Migrating data back from PostgreSQL to SQLite...
# [fix] Restoring monolith database configuration...
# [cleanup] Removing service directory and Docker configuration...
# [sweep] Removing proxy routes from monolith API...
# [config] Updating Nx workspace configuration...
# PASS: orders service retracted successfully!
```

The retraction process reverses each extraction step: migrates data back to the shared database, restores entities to the monolith's ORM configuration, removes the service directory and Docker artifacts, eliminates proxy routes, and updates the extraction manifest. This bidirectional capability treats extraction as a reversible deployment decision rather than a one-way architectural commitment.

Connection to Service Weaver's Design Philosophy

The extract/retract tooling draws heavily from Google's Service Weaver project [GHEMAWAT2023WEAVE which treats distribution as a deployment concern rather than a development concern. Service Weaver applications are written as if they were single-process programs; the runtime decides which components to co-locate and which to distribute based on a deployment configuration file.

The extract command adopts this philosophy for modular monoliths: modules are designed with clean boundaries and infrastructure abstraction (G1–G3), but their deployment topology (monolith vs. service) is determined by the extraction manifest. When a module is marked as `extracted`, the tooling automatically provisions the infrastructure (database, message broker, service runtime) required for independent deployment. When a module is marked as `monolith`, the same code runs in-process with shared infrastructure.

This approach provides several benefits aligned with Service Weaver's design goals:

- *Development/production parity:* Developers can run the entire system as a monolith for local development, while production deploys modules as independent services based on scaling needs.
- *Gradual extraction:* Teams can extract modules incrementally (orders first, then inventory, then payments) based on operational requirements rather than all-or-nothing migration events.
- *Operational flexibility:* The same module can be extracted during peak traffic peri-

ods and retracted during maintenance windows, providing deployment-level scaling strategies.

- *Testing across topologies:* Integration tests can verify module behavior in both monolith and service deployment modes, ensuring that extraction preserves correctness.

Unlike Service Weaver, which operates at the Go component level, the extract command operates at the domain module level and focuses on systems that have already achieved migration readiness through the practices described in G4. This makes it complementary to Service Weaver's approach: where Service Weaver handles distribution automatically, the extract command requires explicit preparation but provides fine-grained control over the extraction process and bidirectional rollback capabilities that are crucial for production adoption.

Implementation Results and Production Readiness

The extract command was implemented and tested against Tiny Store's orders module, producing the following operational results:

Extraction Step	**Duration**	**Result**
Entity discovery	<1s	1 entity found (OrderEntity → orders table)
Database migration	5s	Dedicated PostgreSQL database created
Service scaffolding	<1s	Complete Express app generated
Monolith updates	<1s	Proxy routes installed
Docker configuration	<1s	Dockerfile and compose services
Nx integration	<1s	Project configs and scripts updated
Verification	3s	Integration tests pass
Total extraction	**<15s**	**Orders running as service**
Retraction (rollback)	8s	Module restored to monolith

The tooling successfully demonstrated that a properly prepared module (following G4's readiness checklist) can transition between monolith and service deployment in seconds rather than weeks. The generated service preserves the module's API contracts, event publishing behavior, and integration patterns, requiring no changes to downstream consumers.

More importantly, the retract command validated the bidirectional nature of the extraction decision. Teams can extract modules during scaling pressure and retract them during consolidation periods, treating deployment topology as a tunable operational parameter rather than a permanent architectural commitment.

This operationalization of G4's principles provides production teams with the confidence to invest in migration readiness knowing that the transition, when it occurs, will be

automated, reversible, and low-risk. The extract command transforms migration readiness from a theoretical architectural property into a practical engineering capability that can be deployed, tested, and operationalized at scale.

Common Failure Modes and Anti-Patterns

- *"We'll fix it during migration" (Deferred Readiness):* Postponing migration preparation until extraction is imminent. At that point, the accumulated coupling (shared tables, implicit transactions, unversioned APIs) makes extraction a multi-month project instead of a multi-day configuration change. G4 treats readiness as a continuous investment, not a phase.
- *Shared database as a hidden integration layer:* Two modules that appear decoupled at the code level but share database tables are coupled at the persistence level. This is the most common and most expensive migration blocker, because it requires data migration, schema splitting, and potentially retroactive event sourcing to disentangle.
- *Cross-module transactions masking distributed failure modes:* Relying on database transactions to maintain consistency across modules hides the failure modes that will surface after extraction. When the inventory reservation and the order creation are in the same transaction, partial failure is impossible. After extraction, partial failure is the default. Teams that have not designed compensating actions discover this in production. Introducing Temporal inside the monolith *before* extraction forces the team to confront these failure modes early, when debugging is still straightforward.
- *Payload coupling without anti-corruption layers:* When downstream modules consume event payloads directly without translation, any change to the upstream module's event schema breaks all consumers simultaneously. This creates a coordination bottleneck that is incompatible with independent deployment, which is one of the primary motivations for extraction.
- *Premature extraction without readiness:* Extracting a module that has not been prepared (shared state, no ACLs, no saga) creates a distributed monolith: the operational complexity of microservices with the coupling characteristics of a monolith. This is strictly worse than the original state [GRAVANIS2021DONT].

Metrics and Verification

G4 metrics assess whether a module is structurally prepared for extraction, complementing G1's boundary metrics, G2's maintainability metrics, and G3's scalability metrics.

- *Data Ownership Clarity (ω_m):* Reused from G3, scoped to the candidate module. $\omega_m = 1$ means the module owns all its tables exclusively. $\omega_m < 1$ indicates shared

tables that block extraction. *Migration meaning:* prerequisite for schema isolation.

- *Anti-Corruption Layer Coverage (γ):* The proportion of cross-module integration points (event listeners, API consumers) that include an explicit translation layer:

$$\gamma = \frac{|I_{\text{ACL}}|}{|I|}$$

where I is the set of all cross-module integration points and I_{ACL} is the subset mediated by an ACL. *Migration meaning:* $\gamma < 1$ indicates integration points where upstream schema changes will break downstream modules. Target: $\gamma = 1$ for extraction candidates.

- *Saga Completeness (σ):* The proportion of cross-module workflows that have explicitly defined compensating actions for each failure mode:

$$\sigma = \frac{|W_{\text{compensated}}|}{|W|}$$

where W is the set of cross-module workflows involving the candidate module. *Migration meaning:* $\sigma < 1$ indicates workflows that rely on transactional guarantees and will exhibit data inconsistency after extraction.

- *Infrastructure Abstraction Coverage (α_m):* Reused from G3, scoped to the candidate module. Measures whether infrastructure dependencies (event bus, database, cache) are accessed through ports. *Migration meaning:* $\alpha_m < 1$ means extraction requires code changes in domain or application logic, not just infrastructure wiring.

- *API Version Coverage (ν):* The proportion of public API endpoints and event contracts that carry explicit version metadata:

$$\nu = \frac{|E_{\text{versioned}}|}{|E|}$$

Migration meaning: $\nu < 1$ means that after extraction, API evolution will require coordinated deployment of all consumers, eliminating the independent deployment benefit that motivated extraction.

- *Composite Migration Readiness Score (μ_m):* A composite indicator for module m:

$$\mu_m = f(\omega_m,\ \gamma_m,\ \sigma_m,\ \alpha_m,\ \nu_m,\ \varepsilon_m)$$

where ε_m is the extraction readiness score from G3. μ_m provides a single dashboard view of how prepared a module is for extraction.

Verification strategy: G4 metrics are assessed before any extraction decision. When G3's scalability spectrum indicates that a module is approaching L3 (selective extraction), the team evaluates μ_m to determine whether the module is ready. If μ_m is below threshold, the team invests in readiness (Steps 1–5 above) before proceeding. This ensures that extraction is a deliberate architectural decision, not a crisis response.

Documentation Guidelines

- *Module Extraction Runbook:* For each module identified as an extraction candidate, maintain a runbook that documents: the current migration readiness score (μ_m), the remaining preparation steps, the extraction procedure (infrastructure changes), and the rollback procedure. This runbook is updated as readiness improves.
- *Saga Documentation:* For each cross-module workflow, document the happy path, the failure modes, and the compensating actions. Include sequence diagrams that show both the in-process flow (current) and the distributed flow (post-extraction). The saga documentation should be testable: each compensating action should have a corresponding integration test.
- *Event Contract Registry:* Maintain a registry of all published event contracts, including version history, schema, and consuming modules. This registry serves as the source of truth for ACL design and for assessing the impact of schema changes.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to support migration readiness should address:

- *Schema ownership analysis:* Identify which modules access which database tables and flag shared-table violations ($\omega < 1$).
- *Event schema registry:* Track event contract versions, detect breaking changes, and validate consumer compatibility.
- *Durable workflow orchestration:* Manage cross-module sagas with built-in retry, timeout, and compensation support. Temporal is recommended for its durable execution model, workflow versioning, and production-grade observability (execution history, pending activities, failure traces). The Temporal Web UI provides real-time saga visualization without custom tooling.
- *Distributed event streaming:* Provide durable, partitioned event delivery with consumer group semantics. Apache Kafka is recommended for its throughput, exactly-once semantics (via idempotent producers and transactional consumers), topic-based partitioning aligned with module boundaries, and mature ecosystem (Schema Registry for event contract validation, Connect for data pipeline integration).

- *Infrastructure port verification:* Detect direct infrastructure access (bypassing ports) in module code and report α_m.
- *Extraction simulation:* Run the candidate module in isolation (separate process, separate database) against the existing integration test suite to verify that extraction would succeed without code changes.

Extraction Readiness Checklist

Table 3 summarizes the key criteria that determine whether a module is ready for extraction. Each criterion should be verified before initiating the extraction process.

Table 3: Extraction readiness checklist for candidate modules

Criterion	Ready	Not Ready
Event-driven communication	Async events via Kafka	Direct method calls
Schema isolation	Own tables, $\omega_m = 1$	Shared tables across modules
Idempotent consumers	Deduplication keys present	No idempotency guarantees
Saga/compensation	Temporal workflows defined	Ad-hoc error handling
ACL in place	Translation layer at boundary	Direct model sharing
Independent CI pipeline	Nx target with scoped tests	Monolith-wide pipeline only

Reference Implementation

All code listings in this section correspond to files in the Tiny Store reference implementation. The key file paths are:

- `libs/shared/infrastructure/src/event-bus/event-bus.factory.ts` — Event publisher port and Kafka adapter
- `libs/shared/infrastructure/src/kafka/kafka.producer.ts` — Kafka producer with trace propagation
- `libs/shared/infrastructure/src/kafka/kafka.consumer.ts` — Kafka consumer with idempotency
- `libs/shared/infrastructure/src/kafka/topics.config.ts` — Centralized topic and consumer group configuration
- `libs/shared/contracts/src/events/order-placed.event.ts` — Versioned event contract with factory
- `libs/modules/orders/src/workflows/order-fulfillment.workflow.ts` — Temporal saga workflow
- `libs/modules/orders/src/workflows/order-fulfillment.activities.ts` — Temporal activity implementations
- `libs/modules/orders/src/acl/payment-gateway.acl.ts` — Anti-corruption layer for payments
- `libs/modules/orders/src/acl/inventory-gateway.acl.ts` — Anti-corruption layer for inventory

Literature Support Commentary

Migration from monoliths to microservices is one of the most extensively studied topics in modern software architecture. Fritzsch et al. [FRITZSCH2019] identify common migration intentions, strategies, and challenges, noting that teams frequently underestimate the coupling embedded in shared databases and implicit co-location assumptions. Abgaz et al. [ABGAZ2023DECOMPOSITION] provide a comprehensive review of decomposition techniques, but observe that most approaches focus on identifying service candidates rather than on preparing the monolith for extraction. Wolfart et al. [WOLFART2021MODERNIZING] propose a modernization roadmap but acknowledge that proactive readiness, designing for extraction before the need arises, remains underexplored.

The anti-corruption layer concept originates in Evans' Domain-Driven Design [EVANS2003DDD], where it serves as a boundary mechanism between bounded contexts with different models. Its application to modular monoliths, as a proactive migration preparation technique rather than a reactive integration fix, is novel in this context.

The saga pattern is well-established in distributed systems literature [RICHARDSON2018MICROSERVI but is rarely discussed as a monolith-internal pattern. G4 argues that sagas should be introduced *before* extraction, while the system is still co-located and easier to debug, rather than after extraction when distributed failure modes make saga design significantly harder. Temporal [TEMPORAL2024] provides the runtime infrastructure for this approach: its durable execution model persists workflow state across process restarts, and its activity abstraction creates a natural boundary where in-process calls can later be swapped for network calls without changing workflow logic. This positions Temporal not merely as a microservices tool, but as a migration readiness enabler that functions within the monolith.

Similarly, Apache Kafka [KAFKA2024] is introduced not as a scaling tool but as a migration readiness mechanism. By publishing domain events to durable, partitioned topics inside the monolith, teams establish the event streaming infrastructure that will survive extraction. Kafka's consumer group model, exactly-once semantics, and schema registry provide production-grade guarantees that the in-memory event bus cannot offer, making the transition from monolith to distributed system a matter of configuration rather than re-architecture.

G4 fills a gap in the literature by treating migration readiness as a continuous design property rather than a migration-phase activity. By combining data ownership (G3), anti-corruption layers (DDD), durable workflow orchestration (Temporal), event streaming (Kafka), and infrastructure abstraction (G3), G4 provides a structured framework for ensuring that extraction, when it occurs, is a low-risk configuration change rather than a high-risk architectural transformation.

Building on the scalability spectrum from G3, the extraction-ready patterns intro-

duced here feed directly into G5 (Deployment Strategy), which operationalizes deployment topology changes.

With extraction-ready module patterns in place, the deployment strategy must evolve to support both the monolithic deployment and any extracted services without disrupting the development workflow.

4.8 G5: Streamline Deployment Strategy

This section presents G5, which focuses on designing a deployment strategy that reflects and reinforces the modular structure of the monolith. In this dissertation, deployment strategy is treated as a progressive concern: the system ships as a single artifact today, but the pipeline is module-aware from the outset, so that producing multiple deployment artifacts, when extraction occurs, is a configuration change rather than a pipeline rewrite. G5 ensures that the monorepo remains the single source of truth throughout the entire scalability spectrum defined in G3, from a unified monolith (L0) through selective extraction (L3).

G5 is the second guideline in the Operational Fit Dimension. While G4 prepared the orders module with production-grade infrastructure (Kafka, Temporal, schema isolation), G5 addresses the deployment mechanics: how does the system that uses this infrastructure actually get built, containerized, and shipped? The guideline bridges the gap between architectural readiness (G4) and operational reality, ensuring that the CI/CD pipeline, container strategy, and infrastructure provisioning are aligned with the modular architecture rather than treating the monolith as an opaque blob.

Intent and Rationale

Deployment strategy in modular monoliths is frequently treated as trivial: "it is a monolith, so ship one artifact." This simplification is accurate at the start, but it creates a hidden cost. When extraction pressure arrives (G3, Level L3), teams discover that their CI/CD pipeline, Dockerfiles, and infrastructure provisioning assume a single-artifact world. Building a second artifact requires restructuring the pipeline, introducing new build targets, and often duplicating configuration. The deployment dimension becomes the bottleneck, not the code.

The root cause is that most deployment pipelines are designed around *deployment units* ("what gets shipped") rather than *module boundaries* ("what changed and what depends on it"). A module-aware pipeline inverts this: it understands the dependency graph, runs only the tests and builds affected by a change, and can produce one or more deployment artifacts from the same monorepo without structural changes to the pipeline itself.

G5 therefore treats deployment as a spectrum that mirrors G3's scalability spectrum. Just as the architecture progresses from vertical optimization (L0) to selective extraction (L3), the deployment strategy progresses from a single artifact to multiple artifacts produced from the same monorepo. The monorepo does not split. The build toolchain does not change. What changes is the number of entry points the pipeline packages into deployable images.

This approach preserves three properties that splitting the repository would compromise:

- *Atomic refactoring:* Cross-module changes (e.g., updating an event contract and all its consumers) are committed atomically in a single pull request, with all affected tests running in the same CI job.
- *Shared toolchain:* Linting, formatting, type checking, and boundary verification (G1) run against the entire workspace, ensuring consistency across modules regardless of how they are deployed.
- *Dependency graph as source of truth:* The Nx project graph encodes the same module boundaries enforced by G1, so the pipeline's understanding of "what changed" is always aligned with the architecture.

Conceptual Overview

Deployment strategy is embedded by designing the CI/CD pipeline around module boundaries rather than deployment units:

- The monorepo (Nx workspace) remains the single source of truth for all modules, shared libraries, and deployment configurations, regardless of how many services are extracted.
- The pipeline understands the module dependency graph and uses it to scope builds, tests, and deployments to affected modules only.
- Containerization is module-aware: each deployable target has its own Dockerfile (or Dockerfile target) that packages only the module and its transitive dependencies.
- Infrastructure dependencies (Kafka, Temporal, databases) are provisioned as part of the deployment configuration from day one, not bolted on during extraction.

Applicability Conditions and Scope

G5 applies to modular monolith systems where:

- The codebase is organized as a monorepo with explicit module boundaries, as established by G1.
- The system uses or plans to use containerized deployment (Docker, OCI images).
- The team seeks to maintain a single repository even as individual modules are extracted into independently deployable services.

G5 does not prescribe specific CI/CD platforms (GitHub Actions, GitLab CI, Jenkins), container orchestrators (Kubernetes, ECS, Cloud Run), or infrastructure-as-code tools (Terraform, Pulumi, Helm). Its scope is limited to the deployment architecture: how the pipeline relates to module boundaries and how deployment artifacts are produced from the monorepo.

Objectives

- *Module-aware CI/CD:* Ensure the pipeline understands the module dependency graph and scopes builds, tests, and deployments to affected modules only.
- *Single-repo, multi-artifact capability:* Enable the monorepo to produce multiple deployment artifacts (Docker images) without splitting the repository or duplicating the build toolchain.
- *Infrastructure-as-code from day one:* Provision production dependencies (Kafka, Temporal, databases) as part of the standard deployment configuration, so that extraction does not require new infrastructure provisioning.
- *Consistent local and production environments:* Ensure that the local development environment (Docker Compose) mirrors the production deployment topology, reducing environment-specific failures.
- *Rollback as a first-class operation:* Design deployments so that any release can be rolled back independently per deployment target, without requiring coordinated rollbacks across all modules.

Key Principles

- *The monorepo never splits:* Repository splitting is one of the most frequently cited reasons for migration failures. When modules move to separate repositories, atomic refactoring becomes impossible, shared library versioning introduces diamond dependency problems, and boundary verification (G1) can no longer run across the full codebase. G5 requires that all modules, whether deployed as a single monolith or as multiple services, remain in the same Nx workspace. The repository boundary and the deployment boundary are independent concerns.
- *The dependency graph drives the pipeline:* Nx's `affected` command computes which projects are impacted by a given change, based on the project dependency graph. G5 uses this as the primary pipeline optimization: only affected modules are built, tested, and deployed. This reduces CI time proportionally to the size of the change rather than the size of the repository, and it ensures that deployment decisions are grounded in actual dependency relationships rather than file-path heuristics.
- *One Dockerfile per deployment target, not per module:* A deployment target is the unit that gets packaged into a container image. In the monolith phase (D0), there is one deployment target: `apps/api`. When the orders module is extracted (D2), a second deployment target appears: `apps/orders-service`. Each target has its own Dockerfile that includes only the target's code and its transitive dependencies from the monorepo. Modules that are not deployment targets (e.g., `libs/modules/inventory`) are included as dependencies of whichever target consumes them.

- *Infrastructure is a deployment dependency, not an afterthought:* G4 introduced Kafka, Temporal, and isolated database schemas. G5 ensures that these are provisioned as part of the deployment configuration: Docker Compose for local development, Kubernetes manifests (or equivalent) for production. When the system is a monolith, all infrastructure runs alongside a single application container. When a module is extracted, the infrastructure remains; only the application topology changes.
- *Deployment frequency reflects module maturity:* Not all modules need the same deployment cadence. The monolith may deploy weekly, while an extracted high-traffic module (e.g., orders) deploys daily or on-demand. G5 supports heterogeneous deployment frequency by allowing each deployment target to have its own release pipeline, triggered independently by changes to its affected scope.

The Deployment Spectrum

G5 defines a three-level deployment spectrum that teams can traverse progressively. Each level builds on the previous one, and progression is driven by operational need rather than architectural ambition.

Table 4: Deployment Spectrum for Modular Monoliths

Level	Strategy	Description	Trigger Condition
D0	Single Artifact	One Docker image from `apps/api`. All modules ship together. Standard monolith deployment.	Default starting point.
D1	Module-Aware CI	Nx `affected` scopes tests and builds to changed modules. Still produces one artifact, but CI is faster and module-scoped.	CI times grow beyond acceptable thresholds due to repo size.
D2	Multi-Artifact	The monorepo produces multiple Docker images (e.g., `api` + `orders-service`). Shared libs remain in the repo. Each image has its own deployment pipeline.	G3 Level L3 triggered for a module; independent deployment cadence required.

Implementation Mechanisms: Tiny Store Deployment

The following mechanisms demonstrate how G5 is applied to Tiny Store, progressing from D0 through D2.

D0: The Monolith Dockerfile

In the baseline state, Tiny Store is deployed as a single Docker image built from the `apps/api` project. The Dockerfile uses a multi-stage build to minimize image size and

leverages Nx's build output.

Listing 39: Monolith Dockerfile with multi-stage build (`apps/api/Dockerfile`)

```
# apps/api/Dockerfile
FROM node:20-alpine AS base
WORKDIR /app

FROM base AS deps
COPY package.json package-lock.json ./
RUN npm ci --production=false

FROM base AS build
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npx nx build api --skip-nx-cache

FROM base AS production
ENV NODE_ENV=production
COPY --from=build /app/apps/api/.next/standalone ./
COPY --from=build /app/apps/api/.next/static ./apps/api/.next/static
COPY --from=build /app/apps/api/public ./apps/api/public
EXPOSE 3000
CMD ["node", "apps/api/server.js"]
```

This single image contains all four bounded contexts (orders, inventory, payments, shipments), the shared infrastructure, and the composition root. It connects to Kafka, Temporal, and the database as external dependencies. Containerization is a day-one requirement, not something to "set up when needed."

D0: Docker Compose for Local Development

The local development environment mirrors production topology. Kafka, Temporal, and PostgreSQL run as containers alongside the application. This ensures that developers work against the same infrastructure from day one, eliminating "works on my machine" failures caused by in-memory substitutes.

Listing 40: Docker Compose: full infrastructure from day one (`docker-compose.yml`)

```
# docker-compose.yml - infrastructure is a day-one requirement
version: '3.8'
services:
  api:
    build: { context: ., dockerfile: apps/api/Dockerfile }
```

```
  ports: ['3000:3000']
  environment:
    DATABASE_URL: postgres://tiny:store@postgres:5432/tinystore
    KAFKA_BROKER: kafka:9092
    TEMPORAL_ADDRESS: temporal:7233
    ORDERS_SERVICE_URL: http://orders-service:3001
  depends_on: [postgres, kafka, temporal]

orders-service:                        # D1: extracted service
  build: { context: ., dockerfile: apps/orders-service/Dockerfile }
  ports: ['3001:3001']
  environment:
    DB_HOST: postgres
    DB_NAME: tinystore_orders
    KAFKA_BROKER: kafka:9092
    TEMPORAL_ADDRESS: temporal:7233
  depends_on: [postgres, kafka, temporal]

postgres:
  image: postgres:16-alpine
  environment: { POSTGRES_DB: tinystore, POSTGRES_USER: tiny,
    POSTGRES_PASSWORD: store }
  volumes: ['pgdata:/var/lib/postgresql/data']

kafka:
  image: confluentinc/cp-kafka:7.6.0
  environment:
    KAFKA_NODE_ID: 1
    KAFKA_PROCESS_ROLES: broker,controller
    KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
    KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092,CONTROLLER://0.0.0.0:9093
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    CLUSTER_ID: MkU3OEVBNTcwNTJENDM2Qg

redis:
  image: redis:7-alpine
  ports: ['6379:6379']

temporal:
  image: temporalio/auto-setup:1.24
  environment: { DB: postgres12, POSTGRES_USER: tiny,
```

```
      POSTGRES_PWD: store, POSTGRES_SEEDS: postgres }
    depends_on: [postgres]

  jaeger:
    image: jaegertracing/all-in-one:1.54
    ports: ['16686:16686', '4318:4318']

  prometheus:
    image: prom/prometheus:v2.50.0
    volumes:
    ['./infra/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml']

  grafana:
    image: grafana/grafana:10.3.0
    volumes: ['./infra/grafana/provisioning:/etc/grafana/provisioning']
    depends_on: [prometheus, jaeger]

volumes:
  pgdata:
```

D1: Module-Aware CI with Nx Affected

As the repository grows, running all tests and builds on every commit becomes wasteful. D1 introduces Nx's `affected` command to scope CI to the modules impacted by a change. The pipeline structure does not change; only the scope of what runs is narrowed.

Listing 41: Module-aware CI pipeline (`.github/workflows/ci.yml`)

```
# .github/workflows/ci.yml
name: CI
on:
  push: { branches: [main, 'feature/**'] }
  pull_request: { branches: [main] }
jobs:
  affected:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with: { fetch-depth: 0 }
      - uses: actions/setup-node@v4
        with: { node-version: 20, cache: npm }
      - uses: actions/cache@v4
```

```
      with:
        path: .nx/cache
        key: nx-${{ runner.os }}-${{ hashFiles('package-lock.json')
}}-${{ github.sha }}
    - run: npm ci

    - name: Derive NX_BASE
      id: nx-base
      run: |
        if [ "${{ github.event_name }}" = "pull_request" ]; then
          echo "NX_BASE=origin/${{ github.base_ref }}" >>
"$GITHUB_OUTPUT"
        else
          echo "NX_BASE=HEAD~1" >> "$GITHUB_OUTPUT"
        fi

    - run: npx nx affected --target=lint --base=${{
steps.nx-base.outputs.NX_BASE }}
    - run: npx nx affected --target=test --base=${{
steps.nx-base.outputs.NX_BASE }}
    - run: npx nx affected --target=build --base=${{
steps.nx-base.outputs.NX_BASE }}
```

The `affected` command uses Nx's project graph, the same graph that encodes G1's module boundaries, to determine which projects are impacted. A change to `libs/modules/orders` triggers tests for orders and for `apps/api` (which depends on orders), but not for inventory, payments, or shipments unless they are also affected. Boundary verification (`test:boundary`) always runs because it is inexpensive and catches architectural regressions regardless of which module changed.

D2: Multi-Artifact Extraction

When G3's scalability spectrum triggers Level L3 for the orders module, the monorepo produces a second deployment artifact. The orders module becomes a standalone application (`apps/orders-service`) with its own Dockerfile, its own entry point, and its own deployment pipeline. The remaining modules continue to ship inside the original `apps/api` image.

Listing 42: Extracted orders service Dockerfile (D2)

```
# apps/orders-service/Dockerfile
FROM node:20-alpine AS builder
WORKDIR /app
```

```
COPY package*.json nx.json tsconfig*.json ./
COPY libs/modules/orders/ libs/modules/orders/
COPY libs/shared/ libs/shared/
COPY apps/orders-service/ apps/orders-service/
RUN npm ci --production=false
RUN npx nx build orders-service --prod

FROM node:20-alpine AS runner
WORKDIR /app
COPY --from=builder /app/dist/apps/orders-service ./
COPY --from=builder /app/node_modules ./node_modules
ENV NODE_ENV=production
EXPOSE 3001
CMD ["node", "main.js"]
```

Key observations:

- The Dockerfile copies only `libs/modules/orders` and `libs/shared`, not other modules. Nx's project boundaries ensure that orders cannot import inventory, payments, or shipments (G1), so the image is self-contained.
- The `orders-service` entry point connects to the same Kafka cluster and Temporal namespace as the monolith. The Temporal workflow code from G4 runs unchanged. The `KafkaEventPublisher` publishes to the same `orders.events` topic.
- The monolith's Docker Compose is extended, not replaced. A new service entry for `orders-service` is added alongside the existing `api` service. Both connect to the same Kafka, Temporal, and PostgreSQL infrastructure.
- The CI pipeline uses Nx to determine which deployment targets are affected by a change. A change to `libs/modules/orders` triggers builds for both `apps/api` and `apps/orders-service`. A change to `libs/modules/inventory` triggers only `apps/api`.

Note that the Docker Compose configuration in Listing 40 already includes the `orders-service` entry alongside the monolith, reflecting Tiny Store's design principle that both deployment topologies (D0 and D1) are supported from inception. The `orders-service` connects to a dedicated database schema (`tinystore_orders`), completing the schema isolation from G4. The `api` service routes order-related requests to `orders-service` via HTTP through the `ORDERS_SERVICE_URL` environment variable. This routing change is confined to the composition root; no domain logic changes.

Kamal: Production Deployment Without Kubernetes

While Docker Compose serves local development and CI, production deployment requires zero-downtime deploys, SSL termination, and infrastructure provisioning. Tiny Store uses Kamal[7], a deployment tool from Basecamp that provisions Docker containers on bare servers via SSH—without requiring Kubernetes. Kamal aligns with G3's progressive scalability philosophy: it provides production-grade deployment at D0 complexity, and supports D1→D2 progression by adding service entries to the same configuration file.

Listing 43: Kamal deployment configuration (`config/deploy.yml`)

```
# config/deploy.yml - Kamal 2 deployment for Tiny Store
service: tiny-store
image: maurcarvalho/tiny-store

servers:
  web:
    hosts:
      - <%= ENV['DEPLOY_HOST'] %>

registry:
  server: ghcr.io
  username: maurcarvalho
  password:
    - KAMAL_REGISTRY_PASSWORD

builder:
  arch: arm64
  dockerfile: apps/api/Dockerfile
  context: .

env:
  clear:
    DATABASE_URL: postgres://tiny:store@tiny-store-postgres:5432/tinystore
    KAFKA_BROKER: tiny-store-kafka:9092
    TEMPORAL_ADDRESS: tiny-store-temporal:7233
    REDIS_URL: redis://tiny-store-redis:6379
    EVENT_TRANSPORT: kafka
    OTEL_SERVICE_NAME: tiny-store
    OTEL_EXPORTER_OTLP_ENDPOINT: http://tiny-store-jaeger:4318

accessories:
```

[7] https://kamal-deploy.org

```
  postgres:
    image: postgres:16-alpine
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "5432:5432"
    env:
      clear:
        POSTGRES_DB: tinystore
        POSTGRES_USER: tiny
        POSTGRES_PASSWORD: store

  redis:
    image: redis:7-alpine
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "6379:6379"

  kafka:
    image: confluentinc/cp-kafka:7.6.0
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "9092:9092"

  temporal:
    image: temporalio/auto-setup:1.24
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "7233:7233"

  jaeger:
    image: jaegertracing/all-in-one:1.54
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "16686:16686"

proxy:
  ssl: true
  host: <%= ENV['APP_HOST'] || 'tiny-store.example.com' %>
  app_port: 3000
  healthcheck:
    path: /api/health
```

Key observations about the Kamal configuration:

- *D0 with full infrastructure:* A single `kamal setup` command provisions Postgres, Redis, Kafka, Temporal, and Jaeger as "accessories" alongside the application container. The entire production-grade stack from Table 1 is deployed with one com-

mand.

- *D0→D1 progression:* When the orders module is extracted, a second `servers` entry is added for `orders-service` with its own Dockerfile and dedicated database. Kamal deploys both containers independently with zero-downtime rolling updates. No Kubernetes manifests, no Helm charts, no cluster management.
- *Contrast with Kubernetes:* Kubernetes provides powerful orchestration but introduces significant operational complexity (control plane management, YAML manifests, ingress controllers, service meshes). For a startup at D0–D1, Kamal provides the essential deployment properties—zero-downtime deploys, SSL via Let's Encrypt, health checks, rollback—at a fraction of the operational overhead. This aligns with G3's principle of proportional intervention: Kubernetes becomes appropriate at D2+ when multiple services require service discovery and autoscaling.
- *Rollback:* `kamal rollback` reverts to the previous container image instantly, making deployment decisions reversible.

4.8.0.1 Production Deployment with Kamal

While Docker Compose serves local development, production deployment requires a tool that manages zero-downtime deploys, SSL termination, rolling restarts, and infrastructure provisioning on remote servers. Tiny Store uses Kamal [KAMAL2024], a deployment tool from 37signals that deploys containerized applications to bare-metal or VPS servers via SSH, without requiring Kubernetes.

Kamal is chosen because it matches the progressive scalability philosophy: deploy the simplest topology that meets current needs, and evolve when evidence warrants it. A single $20/month Hetzner VPS running Kamal can serve the monolith with all its infrastructure (Postgres, Kafka, Temporal, Redis, Jaeger), while a comparable managed Kubernetes setup would cost $200+/month before accounting for operational complexity. When scaling pressure arrives (G3, Level L3), Kamal scales horizontally by adding servers to the configuration—without changing the deployment tool.

D0: Monolith deployment. The primary Kamal configuration (`config/deploy.yml`) deploys the monolith with all infrastructure managed as Kamal *accessories*:

Listing 44: Kamal D0 configuration (`config/deploy.yml`, trimmed)

```
# config/deploy.yml - Kamal 2 deployment
service: tiny-store
image: maurcarvalho/tiny-store

servers:
  web:
```

```
    hosts: [<%= ENV['DEPLOY_HOST'] %>]

builder:
  dockerfile: apps/api/Dockerfile
  context: .

env:
  clear:
    DATABASE_URL: postgres://tiny:store@tiny-store-postgres:5432/tinystore
    KAFKA_BROKER: tiny-store-kafka:9092
    TEMPORAL_ADDRESS: tiny-store-temporal:7233
    REDIS_URL: redis://tiny-store-redis:6379
    EVENT_TRANSPORT: kafka

accessories:
  postgres:
    image: postgres:16-alpine
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "5432:5432"
    directories: [data:/var/lib/postgresql/data]
  redis:
    image: redis:7-alpine
    host: <%= ENV['DEPLOY_HOST'] %>
  kafka:
    image: confluentinc/cp-kafka:7.6.0
    host: <%= ENV['DEPLOY_HOST'] %>
    port: "9092:9092"
  temporal:
    image: temporalio/auto-setup:1.24
    host: <%= ENV['DEPLOY_HOST'] %>
  jaeger:
    image: jaegertracing/all-in-one:1.54
    host: <%= ENV['DEPLOY_HOST'] %>

proxy:
  ssl: true
  host: <%= ENV['APP_HOST'] %>
  app_port: 3000
  healthcheck: { path: /api/health }
```

The accessories pattern is central to progressive deployment: Postgres, Kafka, Tem-

poral, Redis, and Jaeger are provisioned by Kamal on the same server as the application. They persist across application deploys (Kamal manages their lifecycle independently), and they are shared across deployment targets when extraction occurs.

D1: Adding the extracted orders service. A separate Kamal destination configuration (`config/deploy.orders-service.yml`) deploys the orders service alongside the monolith. Traefik, Kamal's built-in reverse proxy, routes requests based on path prefix:

Listing 45: Kamal D1 configuration for orders service (`config/deploy.orders-service.yml`, trimmed)

```
# config/deploy.orders-service.yml
service: tiny-store-orders
image: maurcarvalho/tiny-store-orders

servers:
  web:
    hosts: [<%= ENV['DEPLOY_HOST'] %>]
    labels:
      traefik.http.routers.tiny-store-orders.rule: >
        Host(`<%= ENV['APP_HOST'] %>`) && PathPrefix(`/api/orders`)
      traefik.http.routers.tiny-store-orders.priority: "200"

builder:
  dockerfile: apps/orders-service/Dockerfile
  context: .

env:
  clear:
    DATABASE_URL: postgres://...?schema=orders  # isolated schema
    KAFKA_BROKER: tiny-store-kafka:9092
    TEMPORAL_ADDRESS: tiny-store-temporal:7233
    EVENT_TRANSPORT: kafka

# Accessories shared with main deploy.yml - no redeclaration needed
```

Deployment scripts. Two shell scripts operationalize the D0 and D1 topologies:

- `bin/deploy` — Deploys the monolith only (`kamal deploy`). First-time usage: `bin/deploy setup` provisions all accessories.
- `bin/deploy-d1` — Deploys both the monolith and the orders service. Executes `kamal deploy` followed by `kamal deploy -d orders-service`.

This progression from D0 to D1 is a deployment topology change, not a rewrite. The same Kafka topics, Temporal workflows, and ACL gateways from G4 operate identically in both topologies. The only difference is whether the orders module runs inside the monolith process or as a separate container behind Traefik's path-based routing.

Common Failure Modes and Anti-Patterns

- *Repository splitting during extraction:* Moving an extracted module to a separate repository breaks atomic refactoring, duplicates shared libraries, and disables workspace-wide boundary verification. The monorepo should remain unified; only deployment targets multiply.
- *"Build everything" CI:* Running all tests and builds on every commit regardless of what changed. This wastes CI resources, slows feedback loops, and creates incentives to skip tests. Module-aware CI (D1) solves this without sacrificing correctness.
- *Infrastructure bolted on during extraction:* Adding Kafka, Temporal, or database provisioning only when a module is extracted. This creates a "big bang" infrastructure change that compounds the risk of the extraction itself. G5 requires infrastructure-as-code from day one (D0).
- *Monolithic Dockerfile for a modular system:* A single Dockerfile that copies the entire repository into the image, including modules that the deployment target does not use. This increases image size, extends build times, and leaks code that should not be present in a specific service's runtime.
- *Coordinated deployments after extraction:* Requiring all deployment targets to be released simultaneously. This negates the independent deployment benefit that motivated extraction. Each target should have its own release pipeline, triggered by changes to its affected scope.

Metrics and Verification

G5 metrics assess whether the deployment strategy supports the modular architecture and enables progressive extraction.

- *CI Scope Efficiency (η_{CI}):* The ratio of modules tested to modules changed, measuring how effectively the pipeline scopes work to affected modules:

$$\eta_{\text{CI}} = \frac{|M_{\text{changed}}| + |M_{\text{transitively\_affected}}|}{|M_{\text{tested}}|}$$

  Deployment meaning: $\eta_{\text{CI}} < 1$ indicates the pipeline tests modules that are not affected by the change (wasted work). $\eta_{\text{CI}} = 1$ means the pipeline scope is optimal.

- *Build Time per Affected Scope (T_{build}):* The wall-clock time to build and test only the affected modules for a given change. *Deployment meaning:* sustained increase in T_{build} for small changes indicates pipeline inefficiency or module dependency bloat.

- *Deployment Artifact Count ($N_{\text{artifacts}}$):* The number of distinct deployment artifacts (Docker images) produced by the pipeline. *Deployment meaning:* $N_{\text{artifacts}} = 1$ at D0; $N_{\text{artifacts}} > 1$ at D2. Growth should track G3 Level L3 extractions.

- *Infrastructure Parity Score (π):* The proportion of production infrastructure dependencies (Kafka, Temporal, databases) that are also present in the local development environment (Docker Compose):

$$\pi = \frac{|I_{\text{local}}|}{|I_{\text{production}}|}$$

  Deployment meaning: $\pi < 1$ indicates environment drift that increases the risk of deployment failures. Target: $\pi = 1$.

- *Independent Deployment Rate (δ):* The proportion of deployments that affect only a single deployment target (no coordinated multi-target releases):

$$\delta = \frac{|D_{\text{independent}}|}{|D|}$$

  Deployment meaning: low δ after extraction indicates that modules are not truly independently deployable, suggesting residual coupling in the deployment pipeline or in shared state.

- *Rollback Success Rate (R):* The proportion of rollback attempts that succeed without requiring coordinated rollbacks of other targets. *Deployment meaning:* low R indicates deployment coupling that G5 is designed to prevent.

Verification strategy: G5 metrics are tracked from D0 onward. Even before extraction, η_{CI} and T_{build} provide early signals about pipeline efficiency. After extraction (D2), δ and R verify that independent deployment is genuinely achievable. A declining δ after extraction indicates that the deployment strategy has not fully decoupled from the monolith, requiring investigation of shared infrastructure or coordinated release dependencies.

Documentation Guidelines

- *Deployment Target Registry:* Maintain a registry of all deployment targets in the monorepo, including: the Nx project that serves as the entry point, the Dockerfile

path, the infrastructure dependencies, and the deployment cadence. This registry is the deployment equivalent of G1's module descriptor.

- *Infrastructure-as-Code:* All infrastructure dependencies (Kafka topics, Temporal namespaces, database schemas) are defined in version-controlled configuration files (Docker Compose, Kubernetes manifests, Terraform modules). Changes to infrastructure follow the same review process as code changes.
- *Extraction Runbook Extension:* G4's extraction runbook is extended with deployment steps: creating the new Dockerfile, adding the deployment target to CI, provisioning the dedicated database, and updating the routing configuration in the monolith's composition root.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to support the deployment strategy should address:

- *Dependency-aware build orchestration:* Compute affected projects from the module dependency graph and scope builds, tests, and deployments accordingly. Nx is the reference implementation in Tiny Store; alternatives include Turborepo and Bazel.
- *Multi-target container builds:* Produce multiple Docker images from a single monorepo, each scoped to a specific deployment target and its transitive dependencies.
- *Local infrastructure provisioning:* Run production dependencies (Kafka, Temporal, PostgreSQL) locally via Docker Compose with configuration parity to production.
- *Pipeline-per-target:* Support independent CI/CD pipelines for each deployment target, triggered by changes to the target's affected scope.
- *Rollback automation:* Enable per-target rollback without requiring coordinated rollbacks of other targets.

Reference Implementation

All code listings in this section correspond to files in the Tiny Store reference implementation:

- `apps/api/Dockerfile` — Monolith multi-stage Docker build
- `docker-compose.yml` — Full local development environment with all infrastructure
- `.github/workflows/ci.yml` — Module-aware CI with Nx affected
- `config/deploy.yml` — Kamal D0 configuration (monolith + accessories)
- `config/deploy.orders-service.yml` — Kamal D1 configuration (extracted orders)
- `bin/deploy` — D0 deployment script
- `bin/deploy-d1` — D1 deployment script (monolith + orders service)
- `nx.json` — Nx workspace configuration with caching and affected defaults

Literature Support Commentary

Deployment strategy in modular monoliths is one of the least explored topics in the academic literature. Microservices research extensively covers CI/CD pipelines, container orchestration, and independent deployment [ARYA2024BEYOND, JOHNSON2024SERVICEWEAVER], but these discussions assume a multi-repository, multi-service baseline. Studies on monoliths [MONTESI2021SLICEABLE, GRAVANIS2021DONT] acknowledge simplified deployment as a benefit but rarely analyze how deployment should evolve as modules are extracted.

The monorepo approach is well-documented in industry practice, particularly at Google [POTVIN2016 and in the Nx ecosystem, but its application to modular monolith architectures, where the repository structure explicitly encodes bounded-context boundaries and the deployment strategy must support progressive extraction, has not been systematically examined in academic literature.

G5 fills this gap by defining a deployment spectrum (D0–D2) that mirrors G3's scalability spectrum, ensuring that deployment strategy evolves proportionally with architectural complexity. By keeping the monorepo as the single source of truth and using the module dependency graph to drive pipeline decisions, G5 preserves the benefits of monolithic simplicity (atomic refactoring, shared toolchain, unified boundary verification) while enabling the operational flexibility of multi-service deployment when extraction is justified.

With G4's extraction-ready modules and the deployment artifacts defined here, G6 (Observability Patterns) closes the feedback loop by providing the monitoring infrastructure that validates each progression.

The deployment pipeline provides the delivery mechanism, but informed scaling decisions require continuous feedback. The final guideline in the Operational Fit dimension introduces the observability infrastructure that closes this loop.

4.9 G6: Introduce Observability Patterns

This section presents G6, which focuses on embedding module-scoped observability into the modular monolith using the same instrumentation, tooling, and data formats that would be used in a distributed deployment. Observability is treated as a prerequisite for evidence-based architectural decisions: without module-level visibility into latency, throughput, error rates, and cross-module interaction patterns, the trigger conditions defined in G3's scalability spectrum and the readiness assessments prescribed by G4 cannot be evaluated. G6 ensures that the system is transparent by construction, not instrumented as an afterthought.

G6 completes the Operational Fit Dimension (G4–G6). While G4 prepared the module for extraction with production-grade infrastructure and G5 established the deployment pipeline, G6 provides the feedback loop that connects the two: the observability data that tells the team *when* to scale (G3), *whether* a module is ready to extract (G4), and *how* the deployment is performing after extraction (G5). Without G6, the scalability spectrum operates on intuition rather than evidence.

Intent and Rationale

Monolithic systems are traditionally treated as observability black boxes. Application Performance Monitoring (APM) tools report aggregate metrics, such as total request latency, overall error rate, and database query time, but these metrics do not distinguish between modules. When the system slows down, the team knows *that* something is slow, but not *which module* is responsible, *which cross-module interaction* is the bottleneck, or *whether the problem is localized* to a single bounded context.

This opacity creates two problems. First, it prevents evidence-based scaling decisions: G3's metrics (Θ_m, $\sigma^2_{L,m}$, $\Delta\Theta$) require module-level attribution that aggregate APM cannot provide. Second, it makes post-extraction comparison impossible: if the team cannot measure module-level performance before extraction, they have no baseline against which to evaluate whether extraction improved or degraded the system.

Microservices solve this problem by default, because each service is a separate process with its own metrics endpoint, its own log stream, and its own trace context. G6 achieves the same visibility inside the monolith by scoping observability to module boundaries. The key insight is that the instrumentation is identical: OpenTelemetry spans, structured log entries with module context, and Prometheus metrics with module labels work the same way whether the module runs in-process or as a separate service. Because this instrumentation is embedded from project inception, the team builds observability muscle, establishes baselines, and validates that the monitoring infrastructure is production-ready as part of normal development. The same dashboards and traces survive extraction unchanged because they were module-scoped from day one.

Conceptual Overview

Module-scoped observability is embedded through three complementary pillars, each applied at the module boundary:

- *Structured logging with module context:* Every log entry carries the originating module name, a correlation ID that tracks requests across module boundaries, and structured fields (JSON) that enable programmatic querying. Log aggregation tools (e.g., Loki, Elasticsearch) can filter by module without code changes after extraction.
- *Module-scoped metrics via OpenTelemetry:* Each module emits latency, throughput, and error rate metrics tagged with the module name. These metrics feed directly into G3's trigger conditions (Θ_m, $\sigma^2_{L,m}$) and G4's extraction readiness assessment. Prometheus scrapes the metrics endpoint; Grafana visualizes them. The same setup works before and after extraction.
- *Distributed tracing inside the monolith:* OpenTelemetry spans are created for each module handler invocation, each Temporal workflow and activity execution, and each Kafka producer/consumer interaction. A single trace visualizes the full order fulfillment flow across module boundaries, making cross-module latency and failure propagation observable. Jaeger (or any OpenTelemetry-compatible backend) displays the traces. After extraction, the trace context propagates over the network automatically via OpenTelemetry's context propagation.

Applicability Conditions and Scope

G6 applies to modular monolith systems where:

- Module boundaries are enforced (G1) and modules are identifiable as distinct units within the codebase.
- The team needs evidence-based input for scaling decisions (G3) and extraction readiness assessments (G4).
- Production infrastructure (Kafka, Temporal) is already in place (G4/G5), providing natural instrumentation points for tracing and metrics.

G6 does not prescribe specific observability vendors or platforms. It is grounded in OpenTelemetry, which is vendor-neutral and supports all major backends (Prometheus, Grafana, Jaeger, Datadog, New Relic). The guideline focuses on what to instrument and how to scope it to modules, not on which SaaS platform to use.

Objectives

- *Module-level attribution:* Every metric, log entry, and trace span is attributable to a specific module, enabling per-module dashboards, alerts, and performance baselines.

- *Cross-module interaction visibility:* Traces show the full request path across module boundaries, including synchronous handler calls, Temporal workflow and activity executions, and Kafka produce/consume cycles.
- *Baseline establishment:* Before any extraction, module-level performance baselines (latency distributions, throughput profiles, error rates) are recorded, enabling before/after comparison when extraction occurs.
- *Alert precision:* Alerts are scoped to modules rather than the entire application, reducing alert fatigue and enabling targeted incident response.
- *Observability survival across extraction:* The same instrumentation, metric names, trace formats, and dashboards work without modification after a module is extracted into a separate service.

Key Principles

- *Instrument at the module boundary, not inside it:* Observability instrumentation is placed at the module's public surface: handler entry points, event publisher calls, event listener invocations, and Temporal activity boundaries. Internal method calls within a module are not instrumented unless they represent distinct domain operations. This keeps instrumentation overhead low and ensures that module-level metrics remain meaningful after extraction (when internal calls become invisible to external traces).
- *Correlation IDs are non-negotiable:* Every request that enters the system receives a correlation ID (or trace ID in OpenTelemetry terms). This ID is propagated to every module invocation, every Kafka message header, every Temporal workflow context, and every log entry. Without correlation IDs, cross-module debugging requires manual timestamp alignment, which is impractical in production. OpenTelemetry's context propagation provides this automatically when spans are created at module boundaries.
- *Metrics feed architectural decisions, not just operational alerts:* G6 metrics are not only for incident response. They are the primary input for G3's scalability spectrum ("which module needs scaling?"), G4's readiness assessment ("is this module performing within expected bounds?"), and G5's deployment decisions ("does this deployment target need independent release cadence?"). Dashboards should be organized by module and by guideline dimension, not only by infrastructure component.
- *Temporal and Kafka provide observability for free:* Temporal's execution history records every workflow start, activity attempt, retry, timeout, and compensation, providing saga-level observability without custom instrumentation. The Temporal Web UI visualizes workflow state in real time. Similarly, Kafka's consumer group lag

metrics (available via JMX or Prometheus exporters) signal backpressure between modules. G6 leverages these built-in capabilities rather than reimplementing them.

- *Same dashboards before and after extraction:* If the observability setup must be redesigned after extraction, the team loses the baseline and the debugging familiarity built during the monolith phase. G6 requires that metric names, trace span names, log structures, and dashboard layouts are designed to be extraction-agnostic. A Grafana dashboard for "Orders Module Latency" should display the same data whether orders runs in-process or as a separate service.

Implementation: Instrumenting the Order Fulfillment Flow

The following implementation demonstrates how G6 is applied to Tiny Store's order fulfillment flow, producing a single trace that spans the API handler, the Temporal workflow, and all four bounded contexts.

Structured Logging with Module Context

Each module uses a shared logger factory that automatically injects the module name and the current correlation ID (extracted from OpenTelemetry's active span context).

Listing 46: Module-scoped structured logger (`libs/shared/infrastructure/src/observability/module-logger.ts`)

```
import { trace, context } from '@opentelemetry/api';

export interface ModuleLogger {
  info(message: string, data?: Record<string, unknown>): void;
  warn(message: string, data?: Record<string, unknown>): void;
  error(message: string, data?: Record<string, unknown>): void;
  debug(message: string, data?: Record<string, unknown>): void;
}

function getTraceInfo(): { trace_id?: string; span_id?: string } {
  const span = trace.getSpan(context.active());
  if (!span) return {};
  const ctx = span.spanContext();
  return { trace_id: ctx.traceId, span_id: ctx.spanId };
}

function log(level: string, moduleName: string,
  message: string, data?: Record<string, unknown>): void {
  const entry = {
    timestamp: new Date().toISOString(),
```

```
    level, module: moduleName, message,
    ...getTraceInfo(), ...data,
  };
  const output = JSON.stringify(entry);
  switch (level) {
    case 'error': console.error(output); break;
    case 'warn':  console.warn(output);  break;
    case 'debug': console.debug(output); break;
    default:      console.log(output);
  }
}

export function getModuleLogger(moduleName: string): ModuleLogger {
  return {
    info:  (message, data?) => log('info',  moduleName, message, data),
    warn:  (message, data?) => log('warn',  moduleName, message, data),
    error: (message, data?) => log('error', moduleName, message, data),
    debug: (message, data?) => log('debug', moduleName, message, data),
  };
}

// Usage: const logger = getModuleLogger('orders');
//        logger.info('Order placed', { orderId, customerId });
```

After extraction, the same logger produces identical JSON output. Log aggregation queries like `module="orders" AND level="error"` work without modification.

Module-Scoped Metrics with OpenTelemetry

Each module registers its own metrics (latency histogram, request counter, error counter) using OpenTelemetry's Metrics API. The module name is a required attribute on every metric, enabling per-module dashboards.

Listing 47: Module-scoped metrics with memoized meter (`libs/shared/infrastructure/src/observability/module-meter.ts`)

```
import { metrics, Meter, Counter, Histogram } from '@opentelemetry/api';

export interface ModuleMetrics {
  meter: Meter;
  requestCounter: Counter;
  errorCounter: Counter;
  latencyHistogram: Histogram;
```

```
}

const moduleMetrics = new Map<string, ModuleMetrics>();

export function getModuleMeter(moduleName: string): ModuleMetrics {
  if (!moduleMetrics.has(moduleName)) {
    const meter = metrics.getMeter(`tiny-store.${moduleName}`);
    const requestCounter = meter.createCounter(`${moduleName}.requests`, {
      description: `Total requests for ${moduleName} module`,
    });
    const errorCounter = meter.createCounter(`${moduleName}.errors`, {
      description: `Total errors for ${moduleName} module`,
    });
    const latencyHistogram =
   meter.createHistogram(`${moduleName}.latency`, {
      description: `Request latency for ${moduleName} module`,
      unit: 'ms',
    });
    moduleMetrics.set(moduleName, {
      meter, requestCounter, errorCounter, latencyHistogram,
    });
  }
  return moduleMetrics.get(moduleName)!;
}
```

These metrics are exported to Prometheus via OpenTelemetry's Prometheus exporter. A Grafana dashboard displays per-module latency distributions, throughput (Θ_m from G3), and error rates. The same dashboard works after extraction because the metric names and attributes are unchanged.

Distributed Tracing Across Module Boundaries

OpenTelemetry spans are created at each module boundary: API handler entry, Temporal workflow start, each Temporal activity invocation, Kafka produce, and Kafka consume. The resulting trace shows the complete order fulfillment flow as a single, connected trace.

Listing 48: Module-scoped tracer with memoization (`libs/shared/infrastructure/src/observability/module-tracer.ts`)

```
import { trace, Tracer } from '@opentelemetry/api';

const tracers = new Map<string, Tracer>();
```

```
export function getModuleTracer(moduleName: string): Tracer {
  if (!tracers.has(moduleName)) {
    tracers.set(moduleName, trace.getTracer(`tiny-store.${moduleName}`));
  }
  return tracers.get(moduleName)!;
}
```

The tracer factory returns a standard OpenTelemetry `Tracer` scoped to the module name. Callers use the standard `startActiveSpan` API to create spans at handler entry points. Trace context propagation across Kafka messages is handled by a dedicated utility:

Listing 49: Kafka trace context propagation via W3C traceparent (`libs/shared/infrastructure/src/observability/kafka-propagation.ts`)

```
import { context, trace, TraceFlags } from '@opentelemetry/api';

const TRACEPARENT_HEADER = 'traceparent';

/** Inject current span context into Kafka message headers. */
export function injectTraceContext(
  headers: Record<string, string>,
): Record<string, string> {
  const span = trace.getSpan(context.active());
  if (!span) return headers;
  const ctx = span.spanContext();
  headers[TRACEPARENT_HEADER] =
    `00-${ctx.traceId}-${ctx.spanId}-${
      (ctx.traceFlags & TraceFlags.SAMPLED) === TraceFlags.SAMPLED
        ? '01' : '00'}`;
  return headers;
}

/** Extract span context from Kafka headers for parent linking. */
export function extractTraceContext(
  headers: Record<string, string | Buffer | undefined>,
): SpanContext | undefined {
  const raw = headers[TRACEPARENT_HEADER];
  if (!raw) return undefined;
  const traceparent = typeof raw === 'string' ? raw :
   raw.toString('utf-8');
  const parts = traceparent.split('-');
  if (parts.length !== 4) return undefined;
  return {
```

```
    traceId: parts[1], spanId: parts[2],
    traceFlags: parts[3] === '01' ? TraceFlags.SAMPLED : TraceFlags.NONE,
    isRemote: true,
  };
}
```

Listing 50: Traced order placement handler using the observability primitives

```
// libs/modules/orders/src/features/place-order/handler.ts
const tracer  = getModuleTracer('orders');
const logger  = getModuleLogger('orders');
const metrics = getModuleMeter('orders');

export async function placeOrderHandler(params: PlaceOrderParams) {
  return tracer.startActiveSpan('orders.placeOrder', async (span) => {
    const start = Date.now();
    try {
      logger.info('Placing order', { customerId: params.customerId });
      const order = Order.create(params);
      await orderRepository.save(order);

      // Publish to Kafka with trace context in headers
      const headers = injectTraceContext({});
      await eventPublisher.publish('orders.events', {
        eventId: generateId(), eventType: 'OrderPlaced',
        version: 1, timestamp: new Date().toISOString(),
        payload: order.toEventPayload(),
      }, headers);

      metrics.requestCounter.add(1, { handler: 'placeOrder' });
      logger.info('Order placed', { orderId: order.id });
      span.setStatus({ code: 0 });
      return order;
    } catch (error) {
      metrics.errorCounter.add(1, { handler: 'placeOrder' });
      span.setStatus({ code: 2, message: (error as Error).message });
      span.recordException(error as Error);
      throw error;
    } finally {
      metrics.latencyHistogram.record(Date.now() - start,
        { handler: 'placeOrder' });
      span.end();
```

```
    }
  });
}
```

When this flow executes, Jaeger displays a trace like:

Listing 51: Example trace in Jaeger (conceptual)

```
Trace: place-order-abc123  (total: 245ms)
|
|-- orders.placeOrder                      [12ms]  module=orders
|   |-- kafka.produce orders.events        [3ms]   module=orders
|
|-- inventory.reserveStock                 [45ms]  module=inventory
|   |-- kafka.consume orders.events        [2ms]   module=inventory
|   |-- kafka.produce inventory.events     [3ms]   module=inventory
|
|-- temporal.orderFulfillment              [180ms] module=orders
|   |-- activity.reserveInventory          [45ms]  module=inventory
|   |-- activity.processPayment            [85ms]  module=payments
|   |-- activity.createShipment            [50ms]  module=shipments
```

This trace is identical whether modules run in-process or as separate services. The only difference after extraction is that network latency appears between spans, which is precisely the information the team needs to evaluate whether extraction impacted performance.

Built-In Observability from Temporal and Kafka

G4 and G5 introduced Temporal and Kafka as production infrastructure. G6 leverages their built-in observability capabilities without additional instrumentation:

- *Temporal Web UI:* Displays every workflow execution, including: workflow ID, current state, activity history (start, complete, fail, retry), compensation events, and execution duration. For the order fulfillment saga, this means the team can inspect any order's fulfillment progress, identify which activity failed, and see whether compensation executed correctly, all without custom logging.
- *Temporal metrics:* Temporal's SDK emits OpenTelemetry-compatible metrics including workflow start rate, activity execution latency, retry count per activity, and workflow failure rate. These metrics are exported to the same Prometheus instance used by the application, enabling unified dashboards.
- *Kafka consumer lag:* The lag between the latest produced offset and the latest consumed offset for each consumer group indicates backpressure between modules.

If the inventory consumer group falls behind on the `orders.events` topic, it signals that inventory processing cannot keep up with order volume, which is a direct input to G3's scalability assessment.

- *Kafka producer metrics:* Record batch size, request latency, and error rate per topic, providing visibility into event publishing performance per module.

OpenTelemetry SDK Bootstrap

The observability infrastructure is initialized once at application startup, before any module code is imported. This ensures that HTTP and Express auto-instrumentation patches are applied globally, and that all module-scoped tracers and meters export to the configured OTLP endpoint (Jaeger for traces, Prometheus for metrics).

Listing 52: OpenTelemetry SDK configuration (`libs/shared/infrastructure/src/observability/otel.config.ts`)

```
import { NodeSDK } from '@opentelemetry/sdk-node';
import { OTLPTraceExporter } from
    '@opentelemetry/exporter-trace-otlp-http';
import { OTLPMetricExporter } from
    '@opentelemetry/exporter-metrics-otlp-http';
import { PeriodicExportingMetricReader } from
    '@opentelemetry/sdk-metrics';
import { HttpInstrumentation } from '@opentelemetry/instrumentation-http';
import { ExpressInstrumentation } from
    '@opentelemetry/instrumentation-express';
import { resourceFromAttributes } from '@opentelemetry/resources';
import { ATTR_SERVICE_NAME, ATTR_SERVICE_VERSION }
  from '@opentelemetry/semantic-conventions';

export function createNodeSDK(config?: Partial<OtelConfig>): NodeSDK {
  const c = { ...createOtelConfig(), ...config };
  return new NodeSDK({
    resource: resourceFromAttributes({
      [ATTR_SERVICE_NAME]: c.serviceName,
      [ATTR_SERVICE_VERSION]: process.env.npm_package_version || '0.0.0',
      'deployment.environment': process.env.NODE_ENV || 'development',
    } as any),
    traceExporter: new OTLPTraceExporter({
      url: `${c.otlpEndpoint}/v1/traces` }),
    metricReader: new PeriodicExportingMetricReader({
      exporter: new OTLPMetricExporter({
        url: `${c.otlpEndpoint}/v1/metrics` }),
```

```
      exportIntervalMillis: c.metricIntervalMs,
    }),
    instrumentations: [
      new HttpInstrumentation(), new ExpressInstrumentation(),
    ],
  });
}
```

Listing 53: Bootstrap entry point (`libs/shared/infrastructure/src/observability/otel.init.ts`)

```
export function initOpenTelemetry(config?: Partial<OtelConfig>): NodeSDK {
  if (sdk) return sdk;
  sdk = createNodeSDK(config);
  sdk.start();
  process.on('SIGTERM', () => sdk?.shutdown().catch(console.error));
  process.on('SIGINT',  () => sdk?.shutdown().catch(console.error));
  console.log('[otel] OpenTelemetry initialized');
  return sdk;
}
```

The `initOpenTelemetry` call is placed at the very top of the application entry point (`apps/api/src/main.ts`), ensuring all subsequent module imports are automatically instrumented.

Module-Scoped Alert Rules

Prometheus alert rules are scoped to modules, enabling targeted incident response. The following rules are defined in `infra/prometheus/alerts.yml`:

Listing 54: Prometheus alert rules scoped to modules (`infra/prometheus/alerts.yml`)

```
groups:
  - name: module_health
    rules:
      - alert: HighErrorRate
        expr: |
          (sum(rate({__name__=~".+_errors_total"}[5m])) by (module)
           / sum(rate({__name__=~".+_requests_total"}[5m])) by (module)
          ) > 0.05
        for: 5m
        labels: { severity: warning }
        annotations:
```

```
      summary: "High error rate in {{ $labels.module }} module"

  - alert: HighP95Latency
    expr: |
      histogram_quantile(0.95,
        sum(rate({__name__=~".+_latency_bucket"}[5m])) by (le, module)
      ) > 500
    for: 5m
    labels: { severity: warning }
    annotations:
      summary: "High P95 latency in {{ $labels.module }} module"

  - alert: KafkaConsumerLagHigh
    expr: kafka_consumer_group_lag > 1000
    for: 5m
    labels: { severity: critical }
    annotations:
      summary: "Kafka consumer lag high for {{ $labels.group }}"
```

These alerts feed directly into G3's trigger conditions: `HighErrorRate` and `HighP95Latency` signal per-module degradation that may warrant an L0 or L1 intervention, while `KafkaConsumerLagHigh` indicates backpressure between modules that may warrant an L1 (async decoupling) or L3 (extraction) response.

Grafana Dashboard: Module Overview

The reference implementation includes a pre-configured Grafana dashboard (`infra/grafana/dashboard` with four panels that provide the observability surface required by G3 and G4:

- *Request Rate per Module:* `sum(rate({__name__=~".+_requests_total"}[5m])) by (module)` — visualizes Θ_m from G3.
- *P95 Latency per Module:* `histogram_quantile(0.95, ...)` — tracks $\sigma^2_{L,m}$ and triggers L0/L1 interventions.
- *Error Rate per Module:* per-module error rate for alert correlation.
- *Kafka Consumer Lag:* consumer group lag by topic, signaling cross-module backpressure.

Because all metrics are tagged with `module` labels from inception, the same dashboard panels display identical data whether the module runs in-process or as an extracted service. The dashboard is provisioned automatically via Grafana's dashboard provisioning, ensuring it is available in every environment from the first deployment.

Common Failure Modes and Anti-Patterns

- *Aggregate-only monitoring (the black box monolith):* Monitoring only application-level metrics (total latency, total errors) without module attribution. When a performance issue arises, the team cannot isolate which module is responsible, leading to unfocused investigation and delayed resolution.
- *Instrumentation after extraction:* Adding observability only after a module is extracted. This eliminates the pre-extraction baseline, making it impossible to determine whether performance changes are caused by the extraction or by unrelated factors. It also means the team learns to use monitoring tools under the pressure of a production incident.
- *Log-only observability:* Relying exclusively on log searching for debugging without traces or metrics. Logs provide point-in-time information but lack the causal chain that traces provide and the statistical patterns that metrics reveal. Module-scoped debugging requires all three pillars.
- *Custom observability frameworks:* Building bespoke logging, metrics, and tracing infrastructure instead of adopting OpenTelemetry. Custom frameworks create vendor lock-in, increase maintenance burden, and are unlikely to integrate with the observability built into Temporal and Kafka. OpenTelemetry's vendor-neutral approach ensures interoperability.
- *Alert fatigue from application-level thresholds:* Setting alert thresholds on aggregate application metrics rather than per-module metrics. When a single module's latency spikes, an application-level threshold may or may not trigger, depending on overall traffic distribution. Module-scoped alerts (e.g., "orders module p99 latency $>$ 200ms") are more precise and actionable.

Metrics and Verification

G6 metrics assess the completeness and effectiveness of the observability implementation across modules.

- *Trace Completeness Rate (τ):* The proportion of cross-module request flows that produce a complete, connected trace (all expected spans present):

$$\tau = \frac{|F_{\text{traced}}|}{|F|}$$

  where F is the set of all cross-module flows and F_{traced} is the subset with complete traces. *Observability meaning:* $\tau < 1$ indicates instrumentation gaps at module boundaries, which will become debugging blind spots after extraction.

- *Metric Coverage per Module (κ_m):* The proportion of module public surface oper-

ations (handlers, event publishers, event listeners) that emit latency, throughput, and error metrics:

$$\kappa_m = \frac{|O^{(m)}_{\text{instrumented}}|}{|O^{(m)}|}$$

Observability meaning: $\kappa_m < 1$ means that some module operations are invisible to the metrics system, preventing complete Θ_m and $\sigma^2_{L,m}$ calculation for G3.

- *Log Module Attribution Rate (λ):* The proportion of log entries that include a `module` field and a `traceId`:

$$\lambda = \frac{|L_{\text{attributed}}|}{|L|}$$

Observability meaning: $\lambda < 1$ indicates log entries that cannot be filtered by module or correlated with traces. Target: $\lambda = 1$.

- *Alert Precision (α_{alert}):* The proportion of triggered alerts that are scoped to a specific module rather than the application as a whole:

$$\alpha_{\text{alert}} = \frac{|A_{\text{module-scoped}}|}{|A|}$$

Observability meaning: low α_{alert} indicates that alerts lack the specificity needed for targeted incident response.

- *Baseline Coverage (β):* The proportion of modules for which a performance baseline (latency distribution, throughput profile, error rate) has been recorded prior to any extraction:

$$\beta = \frac{|M_{\text{baselined}}|}{|M|}$$

Observability meaning: $\beta < 1$ before extraction means the team cannot evaluate extraction impact for some modules. Target: $\beta = 1$ before any L3 transition.

- *Mean Time to Detect (MTTD):* The average time between a module-level performance degradation occurring and an alert firing. This is the primary operational effectiveness metric. *Observability meaning:* decreasing MTTD over time indicates improving observability maturity.

Verification strategy: G6 metrics are assessed as part of the extraction readiness review (G4). Before any module is extracted, τ, κ_m, λ, and β must meet threshold values. After extraction, the same metrics are compared against the pre-extraction baseline to verify that observability survived the topology change. A drop in τ after extraction indicates that trace context propagation is failing across the network boundary, requiring investigation of OpenTelemetry context propagation configuration.

Documentation Guidelines

- *Observability Runbook:* For each module, document the available dashboards, the metric names and their meaning, the expected alert thresholds, and the debugging workflow ("when this alert fires, check this dashboard, then look at this trace"). This runbook is part of the module's operational documentation and should be reviewed during G4's extraction readiness assessment.
- *Dashboard Registry:* Maintain a registry of Grafana dashboards organized by module and by guideline dimension. Each dashboard should include: the G3 scalability metrics (Θ_m, $\sigma^2_{L,m}$, $\Delta\Theta$), the G4 readiness indicators, and the G6 observability health metrics (τ, κ_m).
- *Performance Baseline Archive:* Before any extraction, record the module's performance baseline (latency percentiles, throughput range, error rate distribution) as a versioned document. This baseline serves as the reference for post-extraction comparison.

Tooling Capabilities Checklist

Any open-source or proprietary tool used to support module-scoped observability should address:

- *Vendor-neutral instrumentation:* OpenTelemetry SDK for traces, metrics, and logs. Vendor-neutral instrumentation ensures portability and avoids lock-in.
- *Metrics collection and visualization:* Prometheus for metrics scraping and storage; Grafana for per-module dashboards. OpenTelemetry's Prometheus exporter bridges the two.
- *Distributed trace collection:* Jaeger (or any OpenTelemetry-compatible backend) for trace visualization. Trace context propagation via OpenTelemetry ensures traces span module boundaries, Kafka messages, and Temporal activities.
- *Log aggregation:* Loki, Elasticsearch, or equivalent for structured log aggregation with support for filtering by module name and trace ID.
- *Workflow observability:* Temporal Web UI for real-time workflow and activity inspection. Temporal's OpenTelemetry integration for workflow-level metrics.
- *Event stream monitoring:* Kafka consumer lag monitoring via JMX exporters or Prometheus Kafka exporter, integrated into per-module Grafana dashboards.

Observability Maturity Levels

Table 5 defines the observability maturity levels that correspond to increasing instrumentation depth within the modular monolith.

Table 5: Observability maturity levels for modular monoliths

Level	Traces	Metrics	Logs	Dashb
O0 (None)	—	—	—	—
O1 (Application)	App-wide	Global counters	Unstructured	Single ov
O2 (Module-scoped)	Per-module spans	Module counters	Structured, tagged	Per-mo
O3 (Cross-boundary)	Cross-module correlation	Θ_m, κ_m	Trace-linked	Per-modul

Literature Support Commentary

Observability is extensively discussed in the microservices literature, where distributed tracing, centralized logging, and metrics collection are considered essential operational requirements [ARYA2024BEYOND, JOHNSON2024SERVICEWEAVER]. However, these practices are rarely advocated for monolithic systems, where the assumption is that a single process does not need distributed observability tools.

This assumption creates a dangerous gap. When a module is extracted, the team must simultaneously learn new monitoring tools, establish baselines, and debug a newly distributed system. G6 eliminates this gap by introducing production-grade observability (OpenTelemetry, Prometheus, Grafana, Jaeger) inside the monolith, scoped to module boundaries. The result is that extraction does not change the observability posture: the same dashboards, the same traces, and the same alerts continue to function, with the addition of network latency visibility between spans.

Montesi et al. [MONTESI2021SLICEABLE] note that monoliths lack comparable observability practices to microservices, treating the system as a "single black box." G6 addresses this directly by making the monolith's internal module interactions as observable as inter-service communication in a distributed system. The key contribution is not the tooling itself, which is well-established, but the practice of introducing it at the module boundary level inside a monolith, creating continuity of observability across the entire scalability spectrum from L0 through L3.

Together with G3's scalability spectrum, G4's extraction patterns, and G5's deployment pipeline, the observability infrastructure completes a self-reinforcing cycle where monitoring evidence drives scaling decisions.

Reference Implementation

The observability infrastructure described in this guideline is fully implemented in the Tiny Store reference implementation. The observability layer resides in `libs/shared/infrastructure/` and comprises six files: `otel.config.ts` and `otel.init.ts` (SDK bootstrap), `module-tracer.ts` (per-module tracer factory), `module-meter.ts` (per-module metrics factory), `module-logger.ts` (structured logger with trace correlation), and `kafka-propagation.ts` (W3C traceparent injection/extraction for Kafka messages). The infrastructure layer (`infra/`) includes

Prometheus alert rules (`prometheus/alerts.yml`) scoped to module labels, and a pre-configured Grafana dashboard (`grafana/dashboards/module-overview.json`) with panels for per-module request rate, P95 latency, error rate, and Kafka consumer lag. The Docker Compose configuration provisions Jaeger, Prometheus, and Grafana alongside the application from the first deployment, ensuring that observability is embedded from project inception rather than retrofitted during extraction. The same dashboards and traces survive extraction unchanged because they were module-scoped from day one.

4.10 Guidelines Deferred to Future Research

The original research design proposed twelve guidelines organized across four dimensions. This dissertation fully develops the first six guidelines, covering the Architectural Design Dimension (G1–G3) and the Operational Fit Dimension (G4–G6). Together, these six guidelines form a complete, self-contained framework: G1 enforces modular boundaries, G2 embeds maintainability, G3 defines the progressive scalability spectrum, G4 prepares modules for extraction with production-grade infrastructure (Kafka, Temporal), G5 establishes the deployment pipeline, and G6 provides module-scoped observability. The Tiny Store reference implementation demonstrates all six guidelines as executable practices.

The remaining six guidelines, spanning the Organizational Alignment Dimension (G7–G9) and the Guideline Orientation Dimension (G10–G12), are deferred to future research. These guidelines address concerns that, while architecturally relevant, require empirical investigation beyond the scope of the current work. This section summarizes the intent of each deferred guideline and its anticipated contribution.

Organizational Alignment Dimension

- **G7: Map Ownership to Teams (Conway's Law).** Focuses on aligning module ownership with team structure, leveraging Conway's Law intentionally so that communication structures reinforce module boundaries rather than degrade them. Future research should investigate how ownership models evolve in startups as teams grow from single-digit to multi-team organizations, and whether explicit ownership assignment reduces boundary erosion over time [VITHARANA2024CHALLENGES].
- **G8: Assess DevOps Maturity.** Addresses the operational readiness of the team to support the infrastructure introduced by G4–G6 (Kafka, Temporal, OpenTelemetry). A team that lacks experience with distributed event streaming or durable workflow orchestration may struggle to operate these systems effectively, regardless of how well the architecture is designed. Future research should develop a maturity model that maps DevOps capabilities to the scalability spectrum levels (L0–L3), helping teams assess whether they are operationally ready for each progression.
- **G9: Treat Onboarding as a First-Class Concern.** Treats developer onboarding as an architectural property. A modular monolith's unified codebase offers onboarding advantages over microservices (one repository, one build, one local environment), but these advantages are realized only when module documentation, test environments, and contribution workflows are deliberately designed. Future research should measure onboarding time across architectural styles and identify which structural properties (module READMEs, guided test fixtures, composition root walkthroughs) have the greatest impact on ramp-up speed [MONTESI2021SLICEABLE].

Guideline Orientation Dimension

- **G10: Favor Actionable Patterns.** Champions reusable implementation artifacts, such as code templates, example repositories, and Architecture Decision Record (ADR) snippets, over abstract principles. The Tiny Store repository already serves this purpose for G1–G6; future research should generalize this approach into a pattern library that covers the full guideline set, validated through practitioner adoption studies [ABGAZ2023DECOMPOSITION].
- **G11: Contextualize to Constraints.** Focuses on adapting architectural decisions to startup-specific constraints such as team size, funding stage, and operational maturity. Future research should develop a constraint assessment framework that maps organizational parameters to recommended guideline configurations, enabling teams to prioritize which guidelines to adopt first based on their specific context [SU2024FROM, PRAKASH2024SYSTEMATIC].
- **G12: Structure Guidance Around Dilemmas.** Advocates framing architectural guidance around trade-off dilemmas rather than prescriptive rules. Future research should catalog the most common dilemmas encountered during modular monolith adoption (e.g., "boundary strictness vs. development velocity," "infrastructure investment vs. team capacity") and provide decision criteria grounded in empirical evidence from practitioner interviews and case studies.

Tooling Evolution: AI-Assisted Modularity Enforcement

Beyond the deferred guidelines, this research identifies two complementary tooling directions that could operationalize the guideline set at the developer workflow level:

1. **AI Coding Agent for Modular Monoliths.** An AI-powered pair programming agent that can be integrated into any existing codebase and provides real-time guidance on module boundary enforcement, dependency direction, scalability-level assessment, and migration readiness. The agent would internalize the guidelines (G1–G12) as its knowledge base and offer contextual suggestions during development, code review, and refactoring. This approach lowers the adoption barrier by embedding architectural guidance directly into the developer's workflow rather than requiring teams to study and memorize the guideline set.
2. **Modularity Framework or Plugin.** An evolution of the AI agent into a language-specific framework or plugin (e.g., a Node.js module, a Ruby gem, or an Elixir library) that provides programmatic tooling for modularity enforcement. Such a package would offer build-time boundary checks, dependency graph visualization, scalability-level metrics computation, and extraction readiness scoring as first-class library features. By packaging the guidelines as executable tooling, the framework would make progressive scalability a reproducible engineering practice rather than

an architectural aspiration.

Both directions represent natural extensions of the Tiny Store reference implementation and the production-grade tooling stack (Kafka, Temporal, OpenTelemetry, Nx) introduced in this dissertation. Their development and validation are deferred to future research.

Research Approach for Deferred Guidelines

The deferred guidelines will follow the same development methodology applied to G1–G6: literature-grounded design, operationalization through Tiny Store as a reference implementation where applicable, metric definition for verification, and practitioner validation through semi-structured interviews. The Organizational Alignment guidelines (G7–G9) will require empirical methods beyond code-level analysis, including team surveys, onboarding time measurements, and organizational case studies. The Guideline Orientation guidelines (G10–G12) will require meta-level validation, assessing whether the guidelines themselves are usable, contextualizable, and effective as decision-support tools.

4.11 Verification Plan

To strengthen the practical and academic relevance of the architectural guidelines proposed in this chapter, a verification phase is planned as the next step in this research. The verification focuses on the six fully developed guidelines (G1–G6), covering the Architectural Design and Operational Fit dimensions. Participants will include software engineering professionals from industry as well as academic researchers in the fields of software engineering and computer science, ensuring a comprehensive view of both practice and research.

All participants will receive a brief informed consent form explaining the purpose of the study, how their data will be used, and their right to withdraw at any time. Interview recordings will be stored on a secure research server with restricted access. Transcripts will be anonymized before any analysis to ensure that no identifying details are exposed. Any survey responses will also be collected anonymously. This approach guarantees that practitioner data are handled in accordance with research ethics standards and Brazilian data protection regulations.

Interview Method

The primary method of verification will consist of semi-structured interviews with selected practitioners. Participants will include professionals with relevant expertise in designing or evolving monolithic and microservices systems or leading software architectural decisions in startups. Each interview will explore the perceived usefulness, feasibility, and completeness of the guidelines as well as any limitations or omissions experts may identify. Feedback will be used to refine the guidelines and inform the development of the deferred guidelines (G7–G12).

Interviews will be conducted remotely and will follow a flexible script structured around four thematic areas:

1. General impressions and relevance of the guideline set (G1–G6)
2. Detailed feedback on the production-grade implementation approach (Kafka, Temporal, OpenTelemetry) and its applicability to real-world startup environments
3. Contextual challenges in applying the guidelines in early-stage, cloud-native applications
4. Assessment of the progressive scalability spectrum (G3) and the deployment spectrum (G5) as decision-support tools

Participants will receive summarized descriptions of the guidelines and access to the Tiny Store reference implementation in advance to facilitate discussion. With participant

consent, all interviews will be recorded and transcribed. Transcripts will be stored securely and later coded using thematic analysis.

Analysis Approach

All expert interview transcripts will be examined to identify recurring themes. The researcher will verify:

1. Viability barriers (e.g., tooling limitations, organizational culture misalignment, infrastructure cost concerns)
2. Viability enablers (e.g., clear boundaries, production-grade tooling, progressive adoption path)
3. Gaps or missing details (e.g., unclear terminology, need for additional examples, missing prerequisites)

Feedback will be categorized according to the four evaluation dimensions defined in Chapter : Architectural Design, Operational Fit, Organizational Alignment, and Guideline Orientation. The researcher will cross-reference these comments against the original literature gaps identified in the systematic review to confirm that the guidelines address underexplored needs and to inform any refinements.

4.12 Timeline and Milestones

- March to April 2026: Recruit six to eight expert practitioners and conduct semi-structured interviews focused on G1–G6 and the Tiny Store reference implementation
- May 2026: Complete thematic coding of interview transcripts and apply revisions to G1–G6
- June 2026: Prepare and distribute optional online survey for broader practitioner feedback
- July to August 2026: Integrate survey results, finalize G1–G6, and document verification findings
- September to December 2026: Begin development and validation of deferred guidelines (G7–G12) based on practitioner feedback and empirical investigation

Conclusion and Future Work

This chapter completes the research proposal by summarizing the contributions developed so far, outlining the forthcoming stages of investigation, and defining the boundaries that shape the work.

4.13 Summary of Contributions

This research proposes a set of architectural guidelines for software startups adopting modular monolith architectures that preserve scalability, maintainability, and internal modularity. The guidelines are organized across four dimensions: Architectural Design, Operational Fit, Organizational Alignment, and Guideline Orientation. Of the twelve guidelines originally proposed, six have been fully developed in this dissertation:

- **G1: Enforce Modular Boundaries** establishes modularity as an enforceable architectural property through explicit declarations, automated boundary verification, and build-time checks. The Tiny Store reference implementation demonstrates G1 as an executable practice with controlled violation exercises and measurable metrics.
- **G2: Embed Maintainability** frames maintainability as a bounded cost of change, operationalized through stable contracts, unidirectional dependency flow, and longitudinal structural metrics that detect architectural drift before it becomes irreversible.
- **G3: Design for Progressive Scalability** introduces the central concept of this dissertation: a four-level scalability spectrum (L0–L3) that enables modular monoliths to absorb growth through proportional, evidence-driven interventions, from vertical optimization through selective extraction, without requiring a wholesale migration to microservices.
- **G4: Promote Migration Readiness** prepares modules for extraction with production-grade infrastructure. Using Apache Kafka for durable event streaming and Temporal for workflow orchestration and saga management, G4 demonstrates that a module can be made extraction-ready while remaining inside the monolith. The orders module case study shows that extraction, when triggered, becomes a deployment topology change rather than an architectural rewrite.
- **G5: Streamline Deployment Strategy** establishes a deployment spectrum (D0–D2) that mirrors G3's scalability spectrum. The monorepo remains the single source of truth; only deployment artifacts multiply. Module-aware CI/CD with Nx, containerization with Docker, and infrastructure-as-code from day one ensure that the deployment pipeline evolves proportionally with the architecture.
- **G6: Introduce Observability Patterns** embeds module-scoped observability using OpenTelemetry, Prometheus, Grafana, and Jaeger inside the monolith, scoped

to module boundaries. The same instrumentation, dashboards, and traces survive extraction unchanged, providing the feedback loop that connects scaling decisions (G3), readiness assessments (G4), and deployment monitoring (G5).

Together, these six guidelines form a self-contained framework supported by a coherent, production-grade technology stack (Nx, KafkaJS, Temporal, Redis/ioredis, BullMQ, TypeORM+PostgreSQL, OpenTelemetry, Jaeger, Prometheus+Grafana, Docker, Kamal, GitHub Actions) and grounded in a **working, tested reference implementation** (Tiny Store) with 127 passing tests, 0 lint errors, 38 atomic commits, validated Docker Compose infrastructure (13 services including PostgreSQL, Kafka, Temporal, Redis, Jaeger, Prometheus, Grafana), and production-ready Kamal deployment configuration. All code listings in G3–G6 are drawn directly from the repository's source files and verified against the running system.

The framework's distinguishing characteristic is that schema isolation, caching, async queues, event-driven communication, and distributed tracing are *architectural requirements from project inception*—not migration prerequisites to be added later. Extraction becomes a deployment topology change, not a codebase rewrite. The system is operationally indistinguishable from a distributed deployment while retaining the simplicity of a single deployable unit.

4.14 Extending the Research Process

The next step is to conduct the verification plan outlined in Chapter . Semi-structured interviews with industry practitioners and academic researchers will confirm, refine, or extend the guidelines and reveal any additional constraints or trade-offs. Insights from these interviews, combined with targeted research in the gray literature, will inform revisions leading to a more actionable and context-informed set of heuristics.

The six deferred guidelines (G7–G12), covering Organizational Alignment and Guideline Orientation, will be developed in a subsequent phase based on practitioner feedback and empirical investigation. These guidelines require methods beyond code-level analysis, including team surveys, onboarding measurements, and organizational case studies, and will be validated through the same interview-based methodology applied to G1–G6.

Beyond the deferred guidelines, this research identifies two tooling directions as future work: (1) an AI-powered coding agent that internalizes the guideline set and acts as a pair programmer, providing real-time modularity guidance when integrated into any existing codebase; and (2) the evolution of that agent into a language-specific framework or plugin (e.g., a Node.js module, a Ruby gem, or an Elixir library) that packages boundary enforcement, scalability-level assessment, and extraction readiness scoring as programmatic tooling. These directions would operationalize the guidelines at the developer workflow

level, lowering adoption barriers and making progressive scalability a reproducible engineering practice.

Through this reflective cycle, the guidelines will transition from literature-derived hypotheses to field-validated tools, ensuring clarity, applicability, and relevance in real-world startup environments.

4.15 Scope and Limitations

The research deliberately focuses on early-stage, cloud-native software startups: contexts where architectural decisions are often made under uncertainty, with limited time, infrastructure, and people. This scope excludes large-scale enterprise systems, highly regulated industries, or long-established software organizations, where architectural dynamics may differ substantially.

The guideline set is not intended to function as a prescriptive framework. Rather, it is structured as a series of modular principles, subject to validation, contextualization, and adaptation. The current work is limited to the Architectural Design and Operational Fit dimensions; the Organizational Alignment and Guideline Orientation dimensions are identified as future research directions.

Additional limitations include:

- While the implementation uses a specific technology stack (Kafka, Temporal, OpenTelemetry, Kamal), the architectural patterns it demonstrates—event-driven communication, durable workflows, module-scoped observability, progressive deployment—are transferable to alternative technologies (e.g., RabbitMQ, AWS Step Functions, Datadog, Kubernetes). The guidelines should be validated with alternative stacks in future work.
- The Tiny Store reference implementation is deliberately small. While this makes the guidelines demonstrable and reproducible, it does not capture the full complexity of a production e-commerce system. Validation with larger, real-world codebases is a necessary extension.
- The availability and diversity of expert participants for the verification phase may constrain the generalizability of findings. This risk will be mitigated through transparent methodology, critical triangulation, and clear scope delimitation.

4.16 Final Remarks

This proposal positions architectural guidance not as a static artifact, but as an evolving construct shaped by both empirical evidence and contextual complexity. The guidelines introduced here reflect the first iteration of that construct. The central argument, that modular monoliths can achieve progressive scalability through deliberate, evidence-driven

architectural interventions without premature distribution, is operationalized through six guidelines, a reference implementation, and a production-grade technology stack.

The next phase of this research will move beyond synthesis into engagement: testing whether these principles can meaningfully support the design and evolution of systems that remain simple in structure, scalable in function, and sustainable in team practice.