Platform-Mediated Pareto-Optimized Multi-Agent Interaction: Technical Architecture

Milestone 1 Architecture Specification v0.1

17 November 2025

Daniele Palombi, Avyay Casheekar, and Richard Mallah*
Center for AI Risk Management & Alignment
{daniele, avyay, richard}@carma.org

*corresponding author

Executive Summary

We present the foundational architecture for a multi-agent runtime and arbitration platform designed to advance research into coordination and implicit cooperation mechanisms among heterogeneous third-party AI agents. This project encompasses the design, development, and benchmarking of a platform that executes AI agents within multi-agent environments offering tool AI services and opportunities for longer-term mutual benefit. The system automatically detects inter-agent interactions and arbitrates disagreements, resource conflicts, and other interdependent decisions across arbitrary domains. The arbitration framework integrates tools from mechanism design, social choice theory, and multiobjective optimization, utilizing the platform's semantic services—which provide agents' runtime context—to enable situationally aware and informed conflict resolution. The central objective is to reduce mutually suboptimal outcomes and overcome cooperation barriers arising from inaccessible trust structures, including insular coalitions and opaque competitive dynamics. By enabling agent creators to vest trust in a shared, neutral execution environment rather than directly in competing agents, the platform expands the feasible space of mutually beneficial and Pareto-efficient outcomes.

This document specifies the complete technical architecture for version 0.1 of the platform. It has a companion theory paper *Platform-Mediated Pareto-Optimized Multi-Agent Interaction: Theoretical Foundations*, which we recommend reading first.

We also plan milestone readouts for future versions 0.5 (Phase 2) and 1.0 (Phase 3) of the platform as well.

In its initial version, we present a coordination platform that enables multiple AI agents to share limited resources through provably optimal arbitration. The platform implements Weighted Proportional Fairness—a mechanism design approach proven in the companion theory document to simultaneously achieve measures of efficiency, fairness, and computational tractability, while remaining resistant to collusion.

A Core Purpose: Trustworthy Coordination for AI Systems

The platform serves as a coordination substrate for multiple AI agents with competing objectives. Its fundamental value is providing mathematical guarantees about fairness and efficiency that agents can trust without requiring trust in each other. Where traditional multi-agent systems rely on strategic interaction hoping fairness emerges, this platform directly optimizes for fairness properties proven in the theory document.

Three properties define the platform's trustworthiness:

1. **Pareto Optimality (Theorem 1):** Every allocation lies on the Pareto frontier—no agent can improve without harming another. Agents can trust they're receiving efficient allocations, not arbitrary ones.

2. **Starvation Protection (Theorems 3, 5):** The logarithmic barrier prevents wealthy or coordinated agents from monopolizing resources. Even agents with zero currency receive meaningful allocations. Participation is individually rational—all agents benefit versus independent operation.

3. **Computational Security (Theorem 2):** Polynomial-time optimization with no exponential worst cases. No adversary can craft inputs triggering exponential computation, making the platform robust against complexity attacks.

System Architecture: Configuration-Driven Agents in Managed Execution

The platform executes agents as hybrid configurations—declarative preferences and goals expressed as data, decision logic implemented as executable code. This hybrid model enables the platform to observe agent preferences directly (enabling direct optimization rather than indirect incentive alignment) while preserving agent autonomy in decision-making.

The architectural flow is:

1. **Agents compute needs** based on current state and goals, expressing ideal and minimum quantities for potentially multiple resource types
2. **Platform batches requests** through an embargo queue (100ms window) to ensure fairness independent of network timing
3. **Contention detection** identifies when aggregate demand exceeds available supply, grouping overlapping conflicts
4. **Arbitration** computes allocations by solving the Weighted Proportional Fairness optimization problem via convex programming
5. **Enforcement** applies allocations atomically with transaction semantics ensuring all-or-nothing success
6. **Priority Economy** tracks currency earned from releases and burned for urgency signaling, creating closed-loop equilibrium dynamics

Key Architectural Decisions Realizing Theoretical Properties

Several design choices translate theory into implementation:

Weighted Proportional Fairness via Exponential Cone Programming: The arbitrator formulates allocation as maximizing $\Sigma_i c_i \cdot \log(\Phi_i(A))$ subject to resource constraints. Using Clarabel's exponential cone solver, this computes exact Pareto-optimal allocations in polynomial time (realizing Theorems 1-2). The logarithmic objective structurally prevents starvation (realizing Theorem 3's collusion resistance).

Immutable State Semantics: Every state transition creates a new version rather than mutating the old. This architectural choice provides: thread-safe reads without locking (enabling scalable observation), trivial transaction rollback (enabling atomic multi-resource operations), complete audit trails (enabling verification), and reproducible execution (enabling formal analysis). Immutability is the foundation for correctness reasoning.

Proactive Arbitration: The platform detects potential conflicts before they cause failures and computes allocations that prevent conflicts. This makes coordination value visible (agents see they receive more through coordination than independently) and eliminates failure recovery complexity (conflicts never occur, so no rollback needed).

Joint Contention Optimization: When resource contentions involve overlapping agent sets, the platform arbitrates them jointly rather than separately. This architectural strategy achieves global Pareto optimality (Property 2.3)—finding beneficial trades across resource types that separate arbitrations miss—at the cost of larger optimization problems. For Milestone 1's scale ($n \leq 20$ agents, $m \leq 10$ resource types), the computational cost is acceptable (worst-case <500ms).

Priority Economy with Burn-Not-Pay: Currency flows in a closed loop where agents earn by releasing resources early (quantity × TimeRemaining × DemandMultiplier) and spend by burning for priority (not transferring to others). This creates self-regulating equilibria without wealth accumulation spirals, maintaining long-term fairness across repeated interactions.

Deadlock-Free Locking Hierarchy: A four-level hierarchy (state transition > arbitration > agent-specific > resource-specific) with strict ordering rules prevents circular waits by construction. The mathematical structure of the hierarchy makes deadlock impossible regardless of contention patterns.

Technical Implementation

The system comprises ten major components: PlatformRuntime (orchestration), EventBus (pub-sub communication), EmbargoQueue (fair batching), ContentionDetector (conflict identification with grouping), ProportionalFairnessArbitrator (optimization via Clarabel), PriorityWeightCalculator (Priority Economy), ResourceManager (enforcement), TransactionManager (atomicity), SafetyMonitor (invariant validation), and AgentExecutionManager (lifecycle management).

The platform will be implemented in Java for strong typing and mature concurrency support, and deployed on a single JVM for Milestone 1 to focus on mechanism correctness without distributed systems complexity. The design provides clear extension points for Phase 2 and

beyond: nonlinear preferences, configurable grouping policies, platform services as resources, learning modules, Byzantine detection, and distributed deployment.

Performance Characteristics

For Milestone 1's target scale:

- **Agents:** Up to 20 concurrent agents with diverse preferences
- **Resources:** Up to 10 resource types (computational, informational, physical, collaborative)
- **Arbitration Latency:** Median 100-250ms, 95th percentile <500ms even for contentions involving all agents and multiple resource types
- **Throughput:** Sustained 50+ arbitrations per second
- **Memory:** Under 1GB including complete event history

The polynomial-time guarantees (Theorem 2) ensure these bounds hold even for adversarially-crafted inputs.

Safety and Verification

Four core invariants are checked before every state transition: resource conservation (quantities preserved), non-negativity (no negative holdings), commitment satisfaction (platform never over-promises), and currency accounting (total currency equals endowments plus minting minus burning). Transaction semantics ensure multi-resource operations apply atomically—either all allocations succeed together or none do, with clean rollback on failure.

Six validation scenarios (theory document Appendix E) test key properties empirically: Pareto improvements from coordination, fairness under scarcity, collusion resistance, Priority Economy equilibration, semantic divergence bounds, and worst-case computational performance.

Extensions to Platform Services

Phase 2 extends the platform to coordinate access to specialized AI services—narrow, composable semantic service capabilities like molecular property prediction, semantic search, or protocol generation—treating them as resource types alongside computational and physical resources. This service integration demonstrates the mechanism's fundamental generality: the same Proportional Fairness optimization, Priority Economy dynamics, and safety guarantees apply whether agents compete for CPU cycles, laboratory equipment time, or credits to invoke specialized AI services. The architecture naturally accommodates this extension through its resource-type-agnostic design, requiring only execution infrastructure (service invocation, backend management) and composition safety monitoring (tracking service chains, detecting concerning patterns), not changes to the core coordination mechanism.

Audience and Scope

This document targets implementation teams building Milestone 1's platform (Milestone 2’s implementation), mechanism designers extending the framework (Phase 2 and beyond), and researchers evaluating the approach or its implications. It assumes familiarity with the theory document's mathematical foundations and focuses on translating theoretical properties into implementable system design.

Table of Contents

1. System Overview

1.1 What We're Building

The platform serves as a coordination substrate for multiple AI agents with competing objectives. Think of it as an operating system for agent coordination—just as an OS mediates access to hardware resources (CPU, memory, disk) for competing applications on a desktop, our platform mediates access to abstract resources (compute units, dataset access, lab equipment time) for competing AI agents. But unlike traditional OS schedulers that use simple priority queues or round-robin allocation, our platform computes allocations that are provably Pareto-optimal and structurally resistant to collusion.

The platform isn't merely a resource manager—it's a mechanism design artifact that implements specific game-theoretic properties through architectural choices. By making agents execute as platform-managed processes (not independent opaque programs), we gain complete visibility into their preferences and behaviors. This visibility enables direct optimization for fairness rather than hoping that fairness emerges from strategic interaction. By using Weighted Proportional Fairness as the allocation mechanism, we get exact Pareto optimality with polynomial-time computation (Theorem 2). By integrating a Priority Economy where agents earn currency through good citizenship (releasing resources) and spend it through burning (not paying), we enable urgency signaling without creating winner-take-all dynamics. By implementing aggressive joint contention grouping, we achieve global Pareto optimality across multiple resource types simultaneously (Property 2.3), discovering beneficial cross-resource trades that separate arbitrations would miss.

The architecture reflects these goals through several key design choices: immutable state semantics for reproducibility and correctness reasoning, proactive arbitration where the platform detects contentions before they cause failures, joint contention grouping that maximizes the scope of Pareto optimization, an embargo queue to prevent latency-based competition, and a locking hierarchy that makes deadlock impossible by construction. Each architectural decision reinforces the properties we need: safety, fairness, efficiency, verifiability.

Aligning extensibility for Phase 2, the platform's resource abstraction extends naturally to specialized AI services—narrow capabilities like semantic paper search, molecular property prediction, or experimental protocol generation. These services become resource types that agents request, the arbitrator allocates fairly, and the Priority Economy prices through scarcity signals. An agent might request `compute_units: 50, semantic_scholar_search_credits: 10, molecular_property_prediction_credits: 100` in a single joint optimization, with the mechanism treating service credits identically to computational resources. This extension demonstrates the architecture's fundamental generality: the mathematics of fair allocation, the logarithmic barrier preventing starvation, and the polynomial-time tractability all apply uniformly whether optimizing over compute cycles or service invocations. The complexity in service integration lies not in mechanism redesign but in execution infrastructure (how agents invoke services, how backends are managed) and composition safety monitoring (tracking whether agents chain services in concerning ways), both of which plug into the existing architecture through well-defined extension points.

1.2 High-Level Architecture

The architecture follows a layered model with clear separation of concerns:

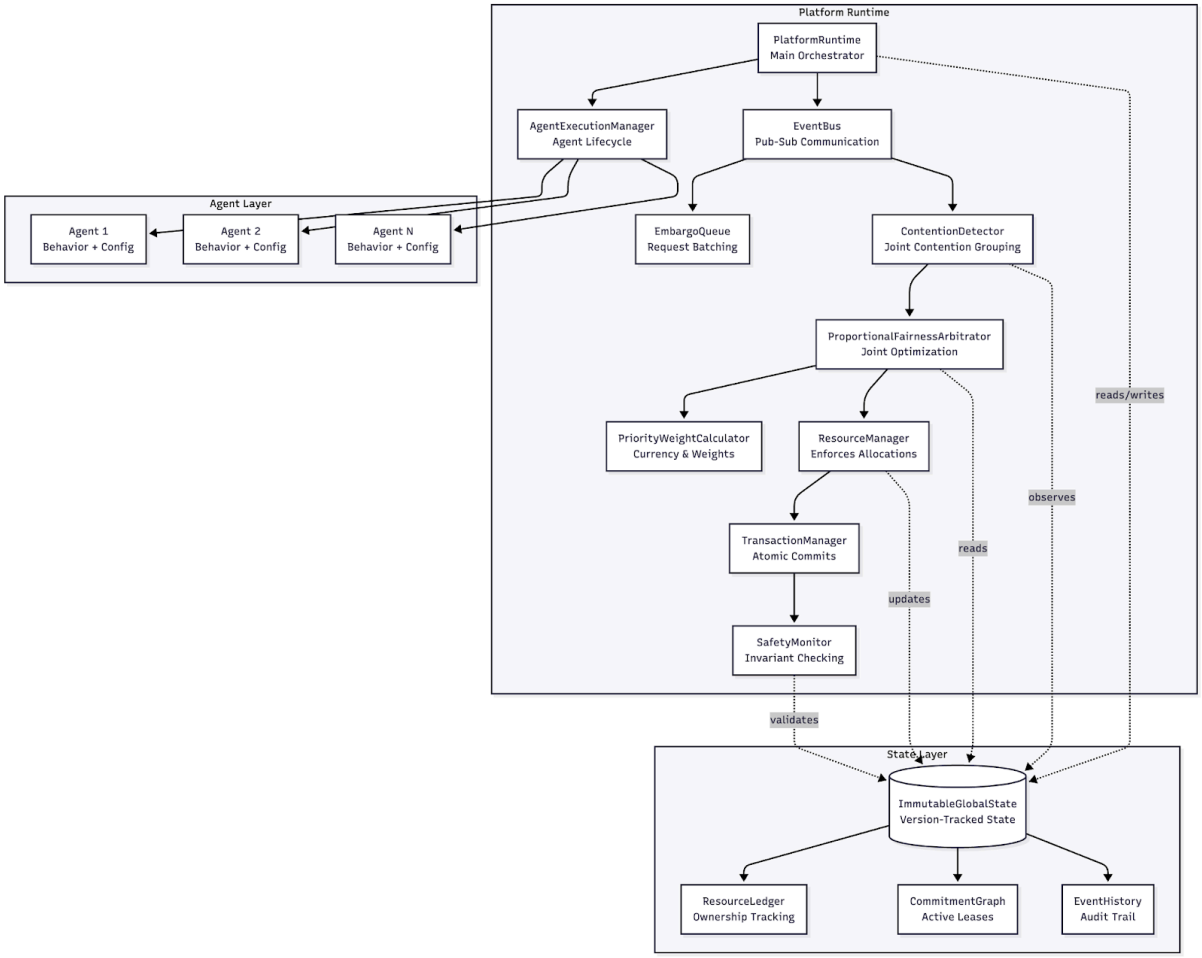

Figure 1. High-Level Platform Architecture

Platform Runtime Layer contains the orchestration logic that manages the entire system lifecycle. The PlatformRuntime component acts as the main event loop, coordinating between subsystems. The EventBus provides publish-subscribe communication that enables sparse activation—components only respond to events they care about, avoiding unnecessary computation. The EmbargoQueue batches near-simultaneous resource requests to prevent agents from gaining advantage through faster network connections or lower latency.

Coordination Logic Layer implements the mechanism design. The ContentionDetector continuously monitors resource requests and implements the aggressive grouping policy—when multiple resource contentions share any agents, they are merged into joint contentions for atomic arbitration. The ProportionalFairnessArbitrator is the mathematical heart of the system, formulating and solving the convex optimization problem that computes globally Pareto-optimal allocations across all contested resource types. The PriorityWeightCalculator manages the Priority Economy, tracking currency balances, computing demand multipliers, and calculating priority weights from currency commitments. The ResourceManager enforces allocations by

updating the ledger, handles currency burning and minting, and tracks resource lease expirations. The TransactionManager provides atomic commit/rollback semantics for joint arbitrations involving multiple resource types and agents.

Agent Management Layer controls agent execution lifecycle. The AgentExecutionManager registers agents from configurations, executes their behavioral code within sandboxes, manages timeouts, and provides them with immutable state snapshots. Agents themselves exist as hybrid configurations—preference functions and goals expressed as declarative data, decision logic implemented as AgentBehavior code.

State Layer maintains the authoritative record of system state. The ImmutableGlobalState serves as the root of all state, with monotonically increasing version numbers that enable optimistic concurrency. The ResourceLedger tracks who owns what resources and how much currency each agent has. The CommitmentGraph records active resource leases with expiration times. The EventHistory provides a complete audit trail of every decision and state transition.

Crosscutting Concerns like the SafetyMonitor validate all state transitions against invariants (resource conservation, non-negativity, commitment satisfaction), rejecting any event that would violate system safety properties.

1.3 Key Architectural Decisions

The table below captures the major decisions that shape this architecture. Each decision involves tradeoffs—we deliberately choose simplicity and provability for Milestone 1, deferring sophistication to later phases when we understand the problem space better.

Table 1. Key Architectural Decisions and Tradeoffs

Dimension	Decision	Rationale	Tradeoff
Language	Java	Strong typing, mature concurrency libraries, excellent threading support, large ecosystem	More verbose than Python; less trendy than Rust; sufficient for our needs
Deployment	Single JVM	Eliminates distributed systems complexity; enables shared memory; simplifies debugging	Limits scale to ~20-50 agents; single point of failure; addressing is out of current scope
State Model	Immutable with structural sharing	Reproducibility, thread-safe reads, event replay, formal reasoning about transitions	Memory overhead (mitigated by structural sharing); slight performance cost

Dimension	Decision	Rationale	Tradeoff
Concurrency	Thread pool + locking hierarchy	Balance simplicity and performance; deadlock prevention by construction	More complex than single-threaded; simpler than lock-free; good sweet spot
Mechanism	Weighted Proportional Fairness	Provably Pareto-optimal (Theorem 1), polynomial-time (Theorem 2), collusion-resistant (Theorem 3)	Requires convex preferences (Phase 1 restriction); exact optimality possible
Contention Handling	Aggressive joint grouping	Global Pareto optimality across resource types (Property 2.3, Corollary 1.2)	Larger optimization problems (n×m variables); worth it for theoretical correctness
Solver	Clarabel (exponential cones)	Specialized for logarithmic objectives; polynomial worst-case; no DoS vulnerability	Adds external dependency; alternative: general nonlinear solver (slower, less secure)
Preferences	Linear over discrete resources	Enables exact optimization; clean theoretical properties; simple to specify	Limited expressiveness; Phase 2 adds nonlinear; sufficient for MVP validation
Arbitration	Proactive (platform-initiated)	Prevents conflicts before they occur; demonstrates coordination value visibly	Platform overhead from monitoring; agents don't control timing; architectural benefit
Currency	Burn-not-pay Priority Economy	Avoids wealth accumulation spirals; signals urgency without buying outcomes; closed-loop dynamics	Novel mechanism (less battle-tested); simple enough to implement; promising properties

Why Aggressive Joint Grouping for Milestone 1: Separate arbitrations of overlapping contentions can only achieve local Pareto optimality within each resource type (Corollary 1.2). By grouping all contentions that share any agents, we ensure global Pareto optimality across the full resource space. This creates larger optimization problems (worst case: all agents × all resource types if one resource is universally contested), but for Milestone 1's scale ($n \leq 20$, $m \leq 10$), the computational cost remains practical (<500ms at 99th percentile). The theoretical correctness benefit—guaranteed global Pareto optimality—outweighs the performance cost. Phase 2 may introduce performance optimizations (parallel separate arbitrations when agent sets are disjoint, approximate methods for very large joint contentions) with formal analysis of when they preserve optimality.

Why Single JVM for Milestone 1: Distributed systems introduce consensus problems, network partitions, clock synchronization, and subtle failure modes. By constraining to single JVM, we can focus on mechanism design and correctness rather than distributed systems engineering. The theoretical properties (Pareto optimality, collusion resistance) are independent of deployment topology—they hold whether we run on one machine or a thousand.

Why Immutable State: Immutability eliminates entire classes of bugs (race conditions from concurrent mutation, inconsistent reads from torn writes, non-reproducible failures from order-dependent effects). The cost is memory overhead, but structural sharing (using persistent data structures like those in Clojure's standard library, available in Java via libraries) keeps this overhead logarithmic. For a system requiring strong correctness guarantees and complete auditability, immutability is the right foundation. It also provides natural transaction semantics for joint arbitrations: we build a complete new state incorporating all changes atomically, then commit the new state or discard it entirely if any step fails.

Why Proactive Arbitration: Traditional multi-agent systems use reactive coordination—conflicts are resolved after they occur, often requiring complex negotiation protocols or rollback mechanisms. Proactive arbitration detects potential conflicts before they happen (by monitoring aggregate demand vs. supply) and computes allocations that prevent the conflicts. This shift from reaction to prevention has profound implications: agents never experience allocation failures, the platform visibly demonstrates coordination value, and we avoid the complexity of failure recovery. The cost is continuous monitoring overhead, but modern processors handle this easily for ~20 agents.

Why Proportional Fairness over VCG: Vickrey-Clarke-Groves (VCG) mechanisms provide dominant-strategy incentive compatibility—agents have incentive to report preferences truthfully. But VCG has critical weaknesses: it's vulnerable to collusion (coordinated misreporting is profitable), it requires solving NP-hard optimization problems (creating DoS vulnerability), and it uses payments that circulate currency (creating wealth accumulation dynamics). Weighted Proportional Fairness addresses all three: collusion is structurally unprofitable (Theorem 3), optimization is polynomial-time (Theorem 2), and currency is burned rather than paid (preventing accumulation). The tradeoff is that we don't get dominant-strategy truthfulness—but with complete platform visibility, we don't need it. The platform reads preferences directly from configurations.

1.4 System Boundaries and Non-Goals

What the Platform Does:

- Executes agent behavioral code in managed, sandboxed environments with complete observability
- Monitors resource requests continuously and detects contentions proactively before failures occur
- Groups overlapping contentions into joint contentions for atomic arbitration (aggressive grouping policy)
- Computes provably globally Pareto-optimal allocations via Weighted Proportional Fairness optimization
- Enforces allocations atomically with transaction semantics (all resources allocated together or none)
- Maintains resource conservation, non-negativity, and commitment satisfaction invariants
- Manages the Priority Economy: agents earn currency by releasing resources, spend by burning for priority
- Provides complete audit trail of all decisions, allocations, and state transitions for verification
- Guarantees starvation protection through logarithmic barrier regardless of wealth disparity

What the Platform Does NOT Do (Milestone 1):

- **Understand semantic meaning** of agent goals beyond observable preferences (Theorem 4 bounds semantic divergence, Theorem 4.1 refines bound under correlation)
- **Learn or predict** future behaviors (Phase 2 adds learning modules as event subscribers)
- **Optimize grouping performance** through selective policies (Phase 2 adds k-hop limits, compatibility matrices, size bounds per Section 7.1.5)
- **Operate in distributed mode** (A future scope may add consensus protocols for multi-node deployment)
- **Handle Byzantine agents** (Milestone 1 assumes non-adversarial; Phase 2 adds detection and quarantine)
- **Support nonlinear preferences** (Phase 1 restriction to linear enables exact polynomial-time optimization)
- **Allocate continuous resources** (Phase 1 uses discrete integers; Phase 2 adds continuous via minor extension)
- **Coordinate specialized AI services** (Phase 2 adds service resource types—credits for semantic search, property prediction, protocol generation—treated identically to computational resources in arbitration but requiring new execution infrastructure for invocation and composition monitoring)
- **Compose across multiple platforms** (future research problem for federated coordination)

These boundaries are deliberate. We build the simplest system that validates core theoretical properties (Pareto optimality, collusion resistance, individual rationality, computational tractability, global optimality). Each non-goal represents a well-defined extension point where future phases add capability without redesigning foundations.

---

2. Architectural Principles and Design Philosophy

The architecture embodies several core principles that guide design decisions and tradeoff resolution. These aren't arbitrary preferences but rather logical consequences of our mission: build a coordination platform with provable safety properties for potentially powerful AI agents.

2.1 Principle 1: Configuration as Executable Artifact

Agents in our system are neither opaque black boxes (as in traditional multi-agent systems) nor fully transparent code (as in operating system processes). They're something in between: **hybrid configurations** that combine declarative data with executable behavior.

The declarative components—preference weights, goal thresholds, initial resource endowments—are expressed as structured data (JSON/YAML). The platform parses these declarations and uses them directly in optimization. This isn't self-reporting (where agents could lie); it's configuration reading. If an agent declares preference weight 0.7 for compute and 0.3 for storage, the platform directly uses these weights in the Proportional Fairness objective function. The agent doesn't get to report different preferences later—what's in the configuration is what the platform optimizes.

The executable components—decision logic, adaptation strategies, prediction heuristics—are implemented as Java classes conforming to the AgentBehavior interface. This code has bounded autonomy: it runs in sandboxed environments with timeouts, resource limits, and no inter-agent communication channels. The behavioral code makes decisions (what resources to request, how much currency to burn, how to adapt to partial allocations), but these decisions are constrained by the platform's mechanism design. An agent can decide to burn 100 currency units for higher priority, but it can't decide to ignore an allocation or steal another agent's resources.

This hybrid model—declarative preferences + executable behavior—enables a sweet spot between restrictiveness and flexibility. The platform can optimize for fairness (using declared preferences) while agents retain decision-making agency (via behavioral code). The configuration serves as a kind of contract: the agent creator commits to preference weights and goals upfront, then the behavioral code implements strategies to achieve those goals within mechanism constraints.

Downstream Implications: Configuration immutability post-submission is essential for this model to work. If agents could modify their configurations after observing outcomes, they could engage in hill-climbing to find exploitative strategies. By freezing configurations at submission time, we simplify the mechanism design problem from dynamic game (with learning and adaptation) to static mechanism (configuration is one-shot commitment). Phase 2 may relax this for experimental exploration, but the production model should maintain immutability.

2.2 Principle 2: Proactive Coordination Over Reactive Resolution

Traditional multi-agent systems operate reactively: agents make requests, conflicts occur, resolution mechanisms kick in. This reactive approach mirrors human organizational structures (escalation hierarchies, conflict mediation) and seems natural. But it has subtle costs: agents experience failures when conflicts occur, resolution requires complex negotiation or rollback, and the value of coordination is demonstrated negatively (by preventing worse outcomes) rather than positively (by achieving better ones).

We invert this model through **proactive arbitration**: the platform continuously monitors aggregate demand against available supply, detects potential contentions before they cause failures, groups overlapping contentions for joint optimization, and computes allocations that prevent conflicts from occurring. Agents never experience allocation failures (unless their minimums are genuinely unsatisfiable, in which case the platform explicitly signals this). Instead, they receive allocations that balance their needs against others' needs across multiple resource types in a provably globally fair way.

This architectural inversion has several benefits. First, it eliminates failure recovery complexity—if conflicts never occur, we don't need mechanisms to roll back or compensate. Second, it makes coordination value visible—agents see that platform-mediated allocation gives them more than independent operation would (Theorem 5's individual rationality). Third, it enables the platform to optimize globally rather than locally—proactive joint arbitration can consider all competing demands across all contested resources simultaneously, discovering beneficial cross-resource trades that reactive resolution or separate per-resource arbitrations would miss (Corollary 1.2).

The cost of proactive arbitration is continuous monitoring overhead. The ContentionDetector must periodically scan all active resource requests, compute aggregate demand for each resource type, compare against available supply, and construct the contention overlap graph for aggressive grouping. For ~20 agents and ~10 resource types, this is ~200 comparisons plus graph traversal every 50-100ms, trivial for modern processors. As we scale to hundreds of agents in contention (Phase 3), this monitoring implementation might become a bottleneck, at which point we can optimize with smarter data structures (spatial indices, incremental computation) or partition monitoring across nodes.

Design Pattern: Proactive arbitration follows the "detect-group-optimize-enforce" pattern rather than the traditional "request-conflict-resolve" pattern. This shift affects everything from event flows (ContentionDetectedEvent triggers joint optimization, not separate ResourceConflictEvents) to agent experience (agents are notified of allocations spanning multiple resource types atomically, not piecemeal allocations for each resource). The pattern is general enough to extend with prediction (Phase 2: predict future contentions and preemptively position resources) or federated coordination (future: multiple platforms detect cross-platform contentions and negotiate fair splits).

2.3 Principle 3: Complete Observability Enables Direct Optimization

Traditional mechanism design assumes agents are strategic actors with private information. The mechanism designer cannot observe agent preferences directly, so must create incentive structures that induce truthful revelation (VCG's dominant-strategy truthfulness) or

game-theoretic equilibria that approximate optimality (auction mechanisms). This adversarial stance—assume agents will lie unless incentivized otherwise—drives much of mechanism design's complexity.

Our architecture makes a different bet: **make agents execute as platform-managed processes**, giving the platform complete observability into configurations, state, and actions. This visibility isn't surveillance in the sense of monitoring after-the-fact; it's architectural integration where the platform is the substrate for agent execution. Agents don't report their preferences to the platform; the platform reads their preference functions directly from their configurations. Agents don't claim to have certain resources; the platform tracks exact resource ownership in the ledger. Agents don't promise to release resources; the platform enforces lease expirations.

This observability enables direct optimization for fairness. Instead of creating clever payment schemes to align incentives (VCG's approach), we can directly optimize the Proportional Fairness objective using agents' true preference functions. Instead of hoping that strategic interaction will produce efficient allocations (market mechanisms), we can compute exact globally Pareto-optimal allocations across all resource types and enforce them atomically. The shift from adversarial mechanism design (align incentives, hope for good outcomes) to cooperative mechanism design (observe preferences, optimize directly) is profound.

The tradeoff is that agents must trust the platform. In traditional mechanisms, agents don't need to trust the mechanism or other agents—the mechanism's incentive properties ensure good outcomes even with mutual distrust. In our model, agents must trust that the platform will execute its algorithm correctly, enforce allocations fairly, and maintain confidentiality of sensitive information. For AI safety applications where we're designing the entire system (platform plus agents), this is acceptable—we can audit the platform, formally verify critical properties, and establish trustworthiness through transparency rather than through game-theoretic cleverness.

Trust Architecture Implications: The platform must be trustworthy to all agent creators, but agents need zero trust in each other. This is similar to how stock exchanges work: traders trust the exchange but not their counterparties. The exchange enforces rules (price-time priority, settlement guarantees) that make counterparty trustworthiness irrelevant. Our platform does the same for resource allocation: by enforcing fair allocations, it makes agent trustworthiness irrelevant while providing stronger guarantees than bilateral trust ever could.

2.4 Principle 4: Immutability as Foundation for Correctness

State management is perhaps the most consequential architectural decision for a concurrent, safety-critical system. Traditional approaches use mutable state with locking—threads acquire locks, modify state in-place, release locks. This works but creates subtle correctness challenges: deadlock (circular lock dependencies), livelock (threads repeatedly back off and retry), race conditions (unsynchronized access), and priority inversion (low-priority thread holds lock needed by high-priority thread). Testing can catch some bugs but cannot prove absence of bugs in concurrent mutable systems.

We choose **immutable state semantics**: every state transition creates a new state version rather than modifying the old one. State transitions are pure functions: `newState = currentState.applyEvent(event)`. The old state remains unchanged, enabling other threads to

continue reading it safely. The new state has a version number one higher, enabling version-based optimistic concurrency. The sequence of states forms an immutable log, enabling event replay for debugging.

Immutability provides several correctness properties essentially for free. Thread-safe reads require no locking—multiple threads can read the same state version simultaneously because it cannot change. Event replay is trivial—apply the same event sequence to the initial state, get the same final state. Rollback is easy—just keep a reference to an earlier state version. Temporal queries work naturally—"what was agent X's resource allocation at time T?" is answered by finding the state version at T. Transaction semantics emerge naturally—build complete new state with all changes, commit atomically or discard entirely on failure. This last property is particularly important for joint arbitrations involving multiple resource types and agents: we apply all allocations together as a single atomic transaction (Property 6.3).

The cost is memory overhead—each state transition creates new data structures. But modern persistent data structures (pioneered by Clojure, available in Java via libraries like PCollections or Vavr) use structural sharing to keep this overhead logarithmic. When we update a map by adding one entry, we don't copy the entire map; we create new nodes for the path from root to the changed entry and share the rest. For our use case (state transitions happen ~10-100 times per second, state size ~10-50MB), structural sharing keeps memory overhead acceptable (~1-2GB for full history over minutes of operation).

Performance Considerations: Immutability is sometimes criticized as slow compared to mutable updates. This is true for tight loops doing many small updates (where allocation overhead dominates), but false for coarse-grained state transitions (where the transition cost dominates). Our state transitions are coarse-grained: we apply one event at a time, and each event represents significant work (arbitration computation, resource transfer, currency update). The overhead of creating new immutable structures is negligible compared to the arbitration computation or event processing logic. In exchange, we get strong correctness guarantees that would require extensive testing to achieve with mutable state.

2.5 Principle 5: Mechanism Properties Through Mathematical Structure

The platform's fairness, efficiency, and security properties don't emerge from careful engineering or extensive testing—they follow from mathematical structure in the mechanism design. This is the power of formal mechanism design: prove theorems about abstract mechanisms, then implement systems that realize those theorems, and the implementations inherit the theorems' guarantees.

Pareto Optimality (Theorem 1) follows from maximizing the product of utilities, which is mathematically equivalent to maximizing the sum of logarithms. The strengthened proof with constructive interpretation shows that there exists no feasible allocation that Pareto-dominates the Proportional Fairness solution. When we implement ProportionalFairnessArbitrator to maximize `Σ_i c_i · log(Φ_i(A))`, we're implementing an algorithm whose output is provably Pareto-optimal. No amount of testing could establish this—tests show that specific instances are Pareto-optimal, but the theorem guarantees that all outputs are Pareto-optimal.

Global Pareto Optimality (Property 2.3, Corollary 1.2) follows from aggressive grouping of contentions. By arbitrating all contentions with overlapping agent sets as a single joint

optimization, we ensure the allocation is Pareto-optimal over the full joint resource space, not merely locally optimal within each resource type. Separate arbitrations would each be locally optimal but could miss beneficial cross-resource trades. The aggressive grouping architecture directly implements this global optimization.

Computational Tractability (Theorem 2) follows from convex problem structure. The logarithmic objective is concave, the constraints are linear (defining a convex polytope), so the problem is convex optimization. Convex problems have polynomial-time algorithms (interior point methods) with no exponential worst-case. When we use Clarabel's exponential cone solver, we're leveraging this convexity guarantee—no adversary can craft agent configurations that trigger exponential solve time. Security through mathematical structure.

Collusion Resistance (Theorem 3, heuristic argument) follows from the logarithmic barrier property. As any agent's utility approaches zero, their log term approaches negative infinity, which dominates the objective regardless of other agents' weights. When coalitions attempt to starve victims, the platform aggressively reallocates to prevent objective collapse. The mechanism fights back not because we programmed special anti-collusion logic, but because the mathematics of the objective function makes starvation self-defeating.

Individual Rationality (Theorem 5) follows from the Pareto improvement property combined with fair distribution of gains. The platform only arbitrates when it can achieve aggregate welfare improvement, and Proportional Fairness distributes improvements fairly (agents with low utility have high marginal value, so receive preferential allocations). Over many arbitrations, this creates positive expected value for all participants. Again, not from engineering but from mathematics.

Architectural Implication: When implementing mechanism-critical components (ProportionalFairnessArbitrator, PriorityWeightCalculator, ContentionDetector with grouping), correctness is paramount. These components embody mathematical guarantees—bugs in their implementation would violate the theorems. We should implement them with maximum clarity (even at cost of minor performance), test them exhaustively (including property-based testing against theorem postconditions), and potentially formally verify critical properties (Phase 3). Less critical components (AgentExecutionManager, EventBus) can be more pragmatic—they affect performance and robustness but not mechanism properties.

2.6 Principle 6: Extensibility Through Abstraction and Interfaces

Milestone 1 deliberately constrains the problem space: linear preferences, discrete resources, single-JVM, non-adversarial agents, aggressive grouping always. These constraints enable clean implementation and thorough validation of core properties. But we know we'll need to relax them in Phases 2-3: nonlinear preferences, continuous resources, distributed execution, Byzantine tolerance, performance-optimized grouping policies.

The architecture must support these extensions without requiring ground-up rewrites. We achieve extensibility through **careful interface design and abstraction boundaries**. Each major subsystem exposes an interface that defines its contract with the rest of the platform, while hiding implementation details behind that interface. When we extend capabilities in Phase 2, we can swap implementations or add new interface implementations without touching the rest of the system.

Example: PreferenceFunction Interface

```
public interface PreferenceFunction {
    double evaluate(ResourceBundle allocation);
    Map<ResourceType, Double> getWeights();
}
```

Milestone 1 implements `LinearPreferenceFunction` where `evaluate()` returns `Σ w_i · r_i`. Phase 2 can add `ConvexPreferenceFunction` (logarithmic utilities, diminishing returns), `ThresholdPreferenceFunction` (value zero below threshold, linear above), or even `LearnedPreferenceFunction` (preferences learned from agent behavior). The arbitrator doesn't care—it works with the interface, calling `evaluate()` on allocations and getting back utility values. As long as new implementations return concave functions (required for convex optimization), the mechanism properties hold.

Example: ArbitrationMechanism Interface

```
public interface ArbitrationMechanism {
    AllocationResult arbitrate(Contention contention, GlobalState state);
}
```

Milestone 1 implements `ProportionalFairnessArbitrator`. Future phases could add `WeightedMaxMinArbitrator` (maximize minimum utility, egalitarian), `NashBargainingArbitrator` (different axiomatic foundation), or even `VCGArbitrator` (for comparison). The rest of the platform doesn't care—it triggers arbitration when contentions occur, receives AllocationResult, and enforces it. We could even support multiple mechanisms simultaneously, letting agent creators choose which mechanism governs their interactions.

Abstraction Boundaries: The platform has several natural abstraction boundaries that enable independent evolution of subsystems:

- **Mechanism abstraction:** Arbitrator interface allows swapping allocation algorithms
- **Solver abstraction:** ConvexSolver interface allows different optimization backends (Clarabel, CVXPY+ECOS, custom interior point method)
- **State abstraction:** GlobalState interface allows different persistence strategies (in-memory, database-backed, distributed)
- **Event abstraction:** Event types can be extended without modifying event bus infrastructure
- **Agent abstraction:** AgentBehavior interface allows arbitrary agent implementations
- **Grouping abstraction:** ContentionGroupingPolicy interface (Phase 2) allows alternative grouping strategies

These boundaries aren't theoretical—they're practical engineering that enables iterative refinement without destabilizing working components. When we discover that Clarabel's exponential cone performance isn't satisfactory (hypothetically), we can implement an alternative ConvexSolver and swap it in without touching arbitration logic. When we add

learning modules (Phase 2), they subscribe to events without modifying event producers. The architecture grows through addition, not modification.

2.7 Principle 7: Global Optimality Through Aggressive Grouping

A subtle but important architectural principle is that we prioritize theoretical correctness (global Pareto optimality) over computational performance (fast arbitration). When contentions overlap in their agent sets—even if the overlap is a single agent—we group them into joint contentions and arbitrate atomically. This aggressive grouping policy ensures we never miss Pareto improvements that exist across resource types (Corollary 1.2).

Why This Matters:

Consider two separate contentions:

- Contention 1: Agents $\{A_1, A_2, A_3\}$ compete for compute_units
- Contention 2: Agents $\{A_2, A_3, A_4\}$ compete for storage_blocks

The overlap is $\{A_2, A_3\}$. With aggressive grouping, we create a joint contention containing agents $\{A_1, A_2, A_3, A_4\}$ and resources {compute, storage}, then solve ONE optimization over the full allocation matrix.

This joint optimization can discover trades like "give more compute to A_1 (who values it at w=0.8) and more storage to A_4 (who values it at w=0.8), while A_2 and A_3 get balanced allocations reflecting their mixed preferences." Separate arbitrations—one for compute, one for storage—cannot discover such trades because they optimize each resource independently.

The cost is larger optimization problems. In worst case, if one resource type (say compute) is requested by all agents, then ALL contentions merge into a single giant joint optimization containing all agents and all contested resource types. For Milestone 1's scale (n=20, m=10), this means up to 200 variables, solvable in <500ms. Worth it for guaranteed global Pareto optimality.

Phase 2 Performance Tradeoffs:

As we scale beyond n=50, aggressive grouping's computational cost may become prohibitive. Section 7.1.5 of the theory document outlines grouping policy optimizations: k-hop transitive depth limits (only group contentions within k hops), resource type compatibility matrices (never group certain resource pairs), size bounds with fallback policies (if joint contention exceeds N agents or M resources, use hierarchical or approximate arbitration). These optimizations trade global Pareto optimality for performance, but all include formal approximation bounds showing how much optimality is sacrificed.

For Milestone 1, we commit unconditionally to aggressive grouping—every contention with ANY overlap is grouped. This maximizes theoretical correctness for initial validation. Once we understand the mechanism thoroughly through Milestone 2's empirical results, Phase 2 can introduce performance optimizations with confidence that we're not breaking properties we haven't yet validated.

2.8 Design Philosophy: Simplicity, Correctness, then Performance

The platform architecture follows a clear priority ordering:

Priority 1: Correctness. The platform must implement the mechanism correctly, maintain safety invariants always, and produce allocations that match theoretical properties. Bugs in arbitration logic would violate Pareto optimality. Bugs in state management would violate resource conservation. Bugs in currency tracking would destabilize the Priority Economy. Bugs in grouping logic would compromise global optimality. Correctness isn't negotiable—we choose architectures and implementations that make correctness easy to establish and verify, even if they're slower or more verbose than alternatives.

Priority 2: Simplicity. Simple systems are easier to understand, test, debug, and extend. We deliberately choose simple designs (single JVM, immutable state, straightforward locking hierarchy, aggressive grouping without optimization) over complex but optimized alternatives (distributed consensus, lock-free data structures, clever caching schemes, selective grouping). Simplicity isn't just aesthetic—it's pragmatic. The simpler the system, the fewer places bugs can hide, the faster we can iterate, and the more confidently we can extend. Premature optimization is the enemy; get correctness and simplicity right first.

Priority 3: Performance. Once correctness and simplicity are established, we optimize for performance where measurements show bottlenecks. The architecture is designed to be *adequate* for Milestone 1 targets (20 agents, 10 resource types, 100 events/second, <200ms arbitration latency for typical contentions, <500ms for worst-case joint contentions). If profiling shows arbitration is the bottleneck, we optimize the solver or formulation. If state transitions are the bottleneck, we optimize immutable data structures or reduce copying. But we don't optimize speculatively—we measure, identify bottlenecks, then optimize specifically.

This priority ordering shapes concrete decisions. We use Java (strong types, clear semantics) over C++ (faster but more complex). We use immutable state (correct by construction) over mutable with locks (faster but error-prone). We use Clarabel (polynomial-time guaranteed) over custom heuristics (potentially faster but no guarantees). We use aggressive grouping (global optimality guaranteed) over selective grouping (faster but might miss trades). These choices prioritize correctness and simplicity, accepting performance costs where necessary.

When Performance Matters: Some performance choices are architectural, not optimizations. Using convex optimization rather than general nonlinear optimization is both a correctness choice (polynomial-time guarantee) and a performance choice (practical speed). Using proactive arbitration rather than reactive resolution is an architectural choice that happens to have good performance (prevents conflict overhead). Aggressive grouping is a correctness choice (global Pareto optimality) that happens to have acceptable performance at Milestone 1 scale. These aren't optimizations—they're fundamental design that shapes system properties. What we avoid is premature micro-optimization (hand-optimizing loops, using lock-free data structures) when correctness and clarity should take precedence.

---

3. Core Components

The platform consists of ten major components, each with clearly defined responsibilities and well-specified interfaces to other components. This section provides detailed specifications for each component, explaining not just what it does but why it's designed the way it is and how it interacts with the broader system.

3.1 Component Overview and Dependency Graph

Before diving into individual components, it's useful to see how they fit together. The diagram below shows the dependency structure—which components depend on which others. The dependency graph is deliberately acyclic (no circular dependencies), which makes the system easier to reason about and test.

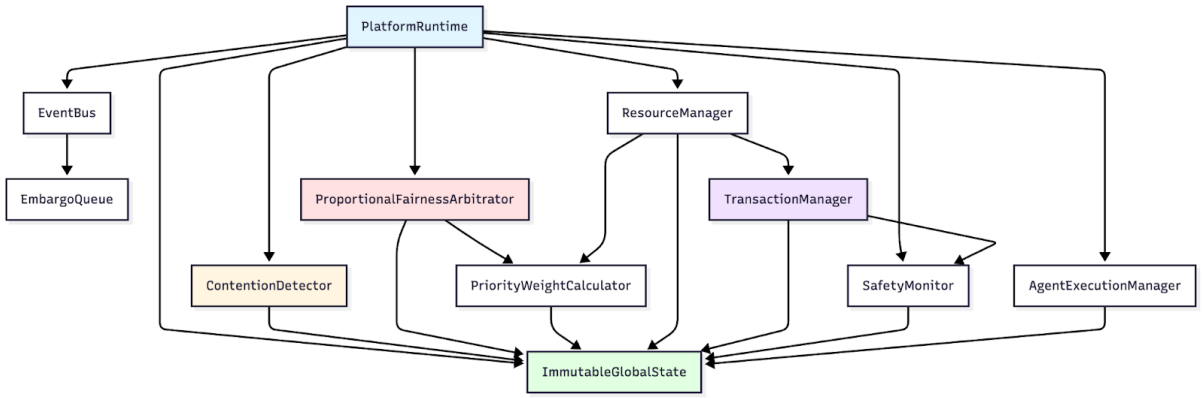

Figure 2. Component Dependency Graph

Dependency Layers:

- **Layer 0 (Foundation):** ImmutableGlobalState—no dependencies, pure data
- **Layer 1 (Basic Services):** EventBus, EmbargoQueue, SafetyMonitor—depend only on state
- **Layer 2 (Domain Logic):** ContentionDetector (with grouping), PriorityWeightCalculator, AgentExecutionManager—depend on state and basic services
- **Layer 3 (Coordination):** TransactionManager (atomic operations), ProportionalFairnessArbitrator (joint optimization), ResourceManager (enforcement)—depend on domain logic
- **Layer 4 (Orchestration):** PlatformRuntime—depends on everything, coordinates execution

This layering enables bottom-up testing: we can test Layer 0 independently, then Layer 1 components with mocked state, then Layer 2 with mocked basic services, and so on. By the time we test PlatformRuntime, all its dependencies have been validated individually.

3.2 PlatformRuntime: The Main Orchestrator

PlatformRuntime is the system's central nervous system—it initializes all subsystems, runs the main event loop, coordinates between components, and manages lifecycle. Think of it as the conductor of an orchestra: it doesn't play instruments (implement core logic), but it ensures everyone plays together in harmony (coordinates execution and state transitions).

Core Responsibilities:

The runtime's primary job is **event-driven orchestration**. It maintains a reference to the current immutable global state and processes events one at a time. Each event represents something that happened (agent requested resources, contention detected, allocation computed) or should happen (enforce allocation, release resources). The runtime applies events to the current state, producing new state versions, and publishes notifications so other components can react.

The runtime also manages **proactive monitoring**. It schedules periodic contention checks (every 50-100ms) where the ContentionDetector scans all active resource requests and performs aggressive grouping to identify joint contentions. When contentions are found, the runtime triggers arbitration. This proactive approach—continuously looking for potential conflicts and grouping them optimally—is what distinguishes our architecture from reactive systems that only respond after problems occur.

Finally, the runtime enforces **safety through pre-transition validation**. Before applying any event, it consults the SafetyMonitor to verify the event would maintain invariants. If validation fails, the event is rejected and the system remains in the current safe state rather than transitioning to an unsafe state. For joint arbitrations, this validation is particularly important since multiple resource types and agents are affected atomically.

Interface Design:

```
public interface PlatformRuntime {
    /**
     * Initialize platform with configuration and initial agents.
     * Validates configuration, loads resource pool, registers agents.
     */
    void initialize(PlatformConfiguration config,
                    Collection<AgentConfiguration> agents);

    /**
     * Start main event processing loop.
     * Blocks until shutdown() is called.
     */
    void run();
```

```
    /**
     * Graceful shutdown: finish processing current event,
     * flush event history, close resources.
     */
    void shutdown();

    /**
     * Get current state version (for monitoring/debugging).
     */
    long getCurrentStateVersion();

    /**
     * Get snapshot of current state (defensive copy).
     */
    ImmutableGlobalState getCurrentState();
}
```

The interface is deliberately minimal—just initialization, execution, and shutdown. Implementation details (event processing logic, state transition mechanics, subsystem coordination) are hidden. This enables different implementations for different deployment scenarios (single-threaded for testing, multi-threaded for production, or even distributed later on) without changing the interface contract.

Event Processing Loop:

The core of PlatformRuntime is its event loop. The general pattern is:

1. **Retrieve next event** from EventBus (blocking if queue empty)
2. **Validate safety** via SafetyMonitor pre-check
3. **Apply event** to current state, producing new state
4. **Commit transition** atomically (update reference to current state via TransactionManager)
5. **Publish notification** (StateTransitionEvent) to inform subscribers
6. **Repeat** until shutdown

This pattern ensures that events are processed serially (one at a time), state transitions are atomic (all-or-nothing), and the system is always in a consistent state (between events, not mid-transition). The serialization might seem inefficient—why not process multiple events concurrently?—but it greatly simplifies reasoning about correctness. Concurrent event processing would require careful analysis of which events can be applied simultaneously (those affecting disjoint state) versus which must be serialized (those that conflict). For Milestone 1's target scale (20 agents, 100 events/second), serial processing is more than adequate. If profiling shows this is a bottleneck (unlikely), Phase 2 can add sophisticated concurrent processing.

Proactive Monitoring:

Parallel to the event loop, the runtime schedules periodic contention checks on a separate thread:

```
scheduledExecutor.scheduleAtFixedRate(
    () -> {
        Set<Contention> contentions =
            contentionDetector.detectAndGroup(getCurrentState(),

agentManager.getActiveAgents());
        for (Contention c : contentions) {
            eventBus.publish(new ContentionDetectedEvent(c));
        }
    },
    0,  // initial delay
    50, // period (milliseconds)
    TimeUnit.MILLISECONDS
);
```

This scheduled task runs every 50ms, checking for contentions and applying aggressive grouping to merge overlapping ones. If found, it publishes ContentionDetectedEvent, which the main event loop will process in due course. The separation between detection (scheduled thread) and processing (main event loop) keeps the architecture clean—detection is periodic monitoring, processing is event-driven reaction.

Coordination Between Subsystems:

PlatformRuntime serves as the integration point where all subsystems come together. It doesn't implement mechanism logic (that's in the arbitrator), resource management (that's in ResourceManager), or agent execution (that's in AgentExecutionManager). Instead, it wires these components together, ensuring data flows correctly between them.

For example, when ContentionDetectedEvent is processed:
1. Runtime extracts the Contention object (which may span multiple resource types and agents)
2. Calls `arbitrator.arbitrate(contention, currentState)` to compute joint allocation
3. Receives AllocationResult with allocations across all resource types and currency burned per agent
4. Calls `transactionManager.executeTransaction()` to apply all changes atomically
5. Within transaction: ResourceManager enforces allocations, burns currency, updates leases
6. TransactionManager validates via SafetyMonitor before commit
7. If validation passes, commits state transition atomically
8. Publishes AllocationEnforcedEvent for agent notification

Each component does its job in isolation, and the runtime orchestrates the sequence. This separation of concerns makes testing easier (mock individual components), debugging simpler (isolate which component caused an issue), and extension cleaner (replace one component without touching others).

3.3 ContentionDetector: Proactive Conflict Identification with Joint Grouping

The ContentionDetector continuously monitors resource requests, looking for situations where aggregate demand exceeds available supply. But it does more than simply identify conflicts—it implements the aggressive grouping policy that ensures global Pareto optimality. When multiple resource contentions share any agents, the detector groups them into joint contentions for atomic arbitration. This grouping transforms separate local optimization problems into a single global optimization that can discover beneficial cross-resource trades.

Detection and Grouping Mechanism:

The detector maintains awareness of all active resource requests through subscription to ResourceRequestBatchEvent. When a batch arrives, it updates its internal view of pending requests. Periodically (triggered by PlatformRuntime's scheduled monitoring), it performs a two-phase process: first detecting which resource types are contested (demand exceeds supply), then grouping these contentions based on agent overlap using graph-based connected components analysis.

Phase 1: Per-Resource Contention Detection

For each resource type, compare aggregate demand against available supply:

```
Map<ResourceType, SingleResourceContention>
detectSingleResourceContentions(
        ImmutableGlobalState state,
        Collection<AgentExecutionContext> agents) {

    Map<ResourceType, SingleResourceContention> contentions = new
HashMap<>();

    for (ResourceType type : state.getResourceTypes()) {
        // Collect requests for this resource type
        List<ResourceRequest> requests = agents.stream()
            .filter(ctx -> ctx.hasActiveRequestFor(type))
            .map(ctx -> ctx.getActiveRequest())
            .collect(Collectors.toList());

        if (requests.isEmpty()) continue;
```

```
        // Compute aggregate demand
        long totalDemand = requests.stream()
            .mapToLong(req -> req.getIdeal(type))
            .sum();

        long availableSupply = state.getResourceLedger()
            .getPoolQuantity(type);

        // Check if contended
        if (totalDemand >= contentionThreshold * availableSupply) {
            Set<AgentId> involvedAgents = requests.stream()
                .map(ResourceRequest::getAgentId)
                .collect(Collectors.toSet());

            contentions.put(type, new SingleResourceContention(
                type,
                involvedAgents,
                requests,
                totalDemand,
                availableSupply
            ));
        }
    }

    return contentions;
}
```

This produces a map of single-resource contentions, each containing the agents requesting that specific resource type. The contentionThreshold (default 1.0) determines sensitivity—we trigger when demand ≥ threshold × supply.

Phase 2: Aggressive Grouping via Graph Analysis

The detector constructs a contention overlap graph where nodes are single-resource contentions and edges connect contentions that share agents. Connected components in this graph become joint contentions:

```
Set<Contention> groupIntoJointContentions(
        Map<ResourceType, SingleResourceContention> singleContentions) {

    // Build overlap graph
    Graph<SingleResourceContention> overlapGraph = new Graph<>();

    // Add all contentions as nodes
    for (SingleResourceContention sc : singleContentions.values()) {
        overlapGraph.addNode(sc);
    }

    // Add edges between contentions that share ANY agents
    List<SingleResourceContention> contentionList =
        new ArrayList<>(singleContentions.values());

    for (int i = 0; i < contentionList.size(); i++) {
        for (int j = i + 1; j < contentionList.size(); j++) {
            SingleResourceContention c1 = contentionList.get(i);
            SingleResourceContention c2 = contentionList.get(j);

            // Check if agent sets overlap
            Set<AgentId> intersection = new
HashSet<>(c1.getInvolvedAgents());
            intersection.retainAll(c2.getInvolvedAgents());

            if (!intersection.isEmpty()) {
                // Share at least one agent - connect them
                overlapGraph.addEdge(c1, c2);
            }
        }
    }

    // Find connected components using BFS/DFS
    Set<Set<SingleResourceContention>> components =
        overlapGraph.findConnectedComponents();

    // Convert each component to JointContention
    Set<Contention> jointContentions = new HashSet<>();
```

```
    for (Set<SingleResourceContention> component : components) {
        jointContentions.add(createJointContention(component));
    }

    return jointContentions;
}

private Contention createJointContention(
        Set<SingleResourceContention> component) {

    // Merge all resource types
    Set<ResourceType> allResourceTypes = component.stream()
        .map(SingleResourceContention::getResourceType)
        .collect(Collectors.toSet());

    // Merge all agents (union across all contentions in component)
    Set<AgentId> allAgents = component.stream()
        .flatMap(sc -> sc.getInvolvedAgents().stream())
        .collect(Collectors.toSet());

    // Collect all requests from all agents
    Map<AgentId, ResourceRequest> requestsByAgent = new HashMap<>();
    for (SingleResourceContention sc : component) {
        for (ResourceRequest req : sc.getRequests()) {
            requestsByAgent.put(req.getAgentId(), req);
        }
    }

    return new Contention(
        allResourceTypes,
        allAgents,
        requestsByAgent,
        component  // Keep original single-resource contentions for
metadata
    );
}
```

Connected Components Algorithm:

The graph traversal for finding connected components uses standard BFS (breadth-first search):

```
Set<Set<Node>> findConnectedComponents() {
    Set<Set<Node>> components = new HashSet<>();
    Set<Node> visited = new HashSet<>();

    for (Node start : allNodes) {
        if (visited.contains(start)) continue;

        // BFS from this start node
        Set<Node> component = new HashSet<>();
        Queue<Node> queue = new LinkedList<>();
        queue.offer(start);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            if (visited.contains(current)) continue;

            visited.add(current);
            component.add(current);

            // Add all neighbors to queue
            for (Node neighbor : getNeighbors(current)) {
                if (!visited.contains(neighbor)) {
                    queue.offer(neighbor);
                }
            }
        }

        components.add(component);
    }

    return components;
}
```

Time complexity: O(V + E) where V is number of single-resource contentions (at most m) and E is number of edges (at most m²/2). For m=10, this is ~100 operations, negligible compared to arbitration time.

Contention Data Structure:

The unified `Contention` class represents both single-resource and joint contentions:

```
public class Contention {
    private final Set<ResourceType> resourceTypes;  // Size 1 for single, 
>1 for joint
    private final Set<AgentId> involvedAgents;
    private final Map<AgentId, ResourceRequest> requestsByAgent;
    private final Map<ResourceType, Long> availableQuantities;
    private final Set<SingleResourceContention> constituentContentions; // 
For tracing

    // Constructors
    public Contention(ResourceType singleType, ...) { ... }  // 
Single-resource
    public Contention(Set<ResourceType> types, ...) { ... }  // Joint

    // Queries
    public boolean isJoint() {
        return resourceTypes.size() > 1;
    }

    public int getProblemSize() {
        return involvedAgents.size() * resourceTypes.size();
    }

    public double getContentionRatio(ResourceType type) {
        long demand = getTotalDemand(type);
        long supply = availableQuantities.get(type);
        return (double) demand / supply;
    }

    public Set<AgentId> getInvolvedAgents() {
        return involvedAgents;
```

```
    }

    public Set<ResourceType> getResourceTypes() {
        return resourceTypes;
    }

    // Utilities for arbitration
    public long getTotalDemand(ResourceType type);
    public long getAvailableSupply(ResourceType type);
    public ResourceRequest getRequestForAgent(AgentId agent);
}
```

The `constituentContentions` field preserves the original single-resource contentions for debugging and analysis (we can trace back which resource-specific contentions were grouped together). The `isJoint()` method enables code to distinguish single-resource from joint contentions where behavior differs.

Detection Algorithm:

The complete detection-and-grouping algorithm:

1. **Scan active requests** (O(n) where n is number of active agents)
2. **Group by resource type** (O(n·m) where m is resource types)
3. **Compute aggregate demands** per type (O(m))
4. **Compare to supply** and create single-resource contentions (O(m))
5. **Build overlap graph** (O(m^2) in worst case—compare all pairs)
6. **Find connected components** (O(m) using BFS)
7. **Merge components into joint contentions** (O(m))

Total complexity: O(n·m + m^2) ≈ O(n·m) when n >> m (typical: 20 agents, 10 resource types). This completes in <1ms, negligible overhead for 50ms detection interval.

Why Aggressive Grouping Works:

The aggressive grouping policy—merge contentions if they share ANY agents—might seem extreme. Why not use a more selective policy (only group contentions with "substantial" overlap, or only group when agents request both resources together)?

The answer is that **any selective policy risks missing Pareto improvements**. Consider:

- Contention 1: Agents {A_1, A_2} want compute
- Contention 2: Agents {A_2, A_3} want storage

If we arbitrate separately:

- Compute arbitration: Optimize A_1 and A_2's compute allocations
- Storage arbitration: Optimize A_2 and A_3's storage allocations
- Both are locally Pareto-optimal (within their resource type)

But there might exist a global improvement: give more compute to A_1 (who values it highly) and more storage to A_3 (who values it highly), with A_2 receiving balanced allocation across both. This trade improves total welfare but requires joint optimization to discover. The single overlapping agent A_2 is the connection that enables the trade.

Aggressive grouping ensures we never miss such improvements. The computational cost—solving one larger problem instead of two smaller problems—is worth it for guaranteed global Pareto optimality (Property 2.3).

Worst-Case Grouping Behavior:

The worst case for aggressive grouping occurs when one resource type is universally desired:

Example:

- Compute contention: All 20 agents request compute_units
- Storage contention: Agents {A_1, A_2, A_3} request storage_blocks
- Dataset contention: Agents {A_4, A_5} request dataset_access
- [7 more resource-specific contentions with various agent subsets]

Because compute contention includes all agents, it overlaps with every other contention. The aggressive grouping merges everything into ONE giant joint contention:

- Agents: All 20 agents
- Resources: All 10 contested resource types
- Problem size: 20 × 10 = 200 variables

For Milestone 1, Clarabel solves 200-variable problems in <500ms (Appendix B complexity analysis). Acceptable for 1-second arbitration budget. The global Pareto optimality achieved is worth this computational cost.

Phase 2's grouping policy optimizations (Section 7.1.5 in theory, Section 11.4 in this document) address scaling beyond n=50 by introducing selective grouping with formal approximation bounds.

Threshold Configuration and Tuning:

The contention threshold determines how sensitive the detector is to over-demand. A threshold of 1.0 (demand ≥ supply) triggers arbitration whenever any oversubscription exists, even by one unit. A threshold of 1.2 (demand ≥ 1.2 × supply) allows 20% oversubscription before triggering, reducing arbitration frequency but allowing more agents to compete before intervention.

For Milestone 1, we use threshold = 1.0 (trigger on any oversubscription) to maximize opportunities for the mechanism to demonstrate value. Every contention becomes an opportunity to compute Pareto-improving allocations across multiple resource types. The cost is higher arbitration frequency (more compute spent solving optimizations, larger joint problems), but for ~20 agents this is acceptable. Phase 2 can experiment with dynamic thresholds that adapt based on arbitration success rate or resource volatility.

Interface Design:

```
public interface ContentionDetector {
    /**
     * Detect contentions and group via aggressive policy.
     * Returns set of Contention objects (each may be joint or
single-resource).
     *
     * @param state Current platform state
     * @param agents Active agent contexts with pending requests
     * @return Set of contentions (grouped by agent overlap)
     */
    Set<Contention> detectAndGroup(
        ImmutableGlobalState state,
        Collection<AgentExecutionContext> agents
    );

    /**
     * Set contention threshold (demand/supply ratio).
     * Default 1.0 = trigger when demand ≥ supply.
     */
    void setContentionThreshold(double threshold);

    /**
     * Get statistics about grouping behavior (for monitoring).
     */
    GroupingStatistics getGroupingStats();
}

public class GroupingStatistics {
    private final int totalSingleResourceContentions;
    private final int totalJointContentions;
    private final int largestJointSize;  // agents × resources
    private final double averageJointSize;
    private final Map<Integer, Integer> sizeDistribution;

    // Getters for monitoring grouping effectiveness
}
```

The `GroupingStatistics` class enables monitoring of how aggressive grouping behaves in practice. We can track:

- How often do contentions get grouped? (Most of the time if agents have diverse resource interests)
- How large do joint contentions get? (Average, maximum)
- What's the size distribution? (Many small joints, few large joints?)

This monitoring informs Phase 2 optimization decisions: if 90% of joint contentions contain only 2-3 resource types, aggressive grouping's overhead is minimal and we should keep it. If 50% contain 8+ resource types, we might benefit from selective grouping.

Example Grouping Scenarios:

Scenario A: Disjoint Agent Sets (No Grouping)

```
Compute contention: Agents {A1, A2, A3}
Storage contention: Agents {A4, A5, A6}
Dataset contention: Agents {A7, A8}

Agent sets are disjoint (no overlap).
Result: Three separate contentions, arbitrated independently in parallel.
Global Pareto optimality still holds (no cross-resource trades possible when agents disjoint).
```

Scenario B: Partial Overlap (Moderate Grouping)

```
Compute contention: Agents {A1, A2, A3}
Storage contention: Agents {A2, A3, A4}
Dataset contention: Agents {A5, A6}

Overlap: {A2, A3} in both compute and storage.
Result:
  - Joint contention 1: {A1, A2, A3, A4} × {compute, storage}
  - Separate contention 2: {A5, A6} × {dataset}
Two arbitrations: one joint (4×2 problem), one single-resource (2×1 problem).
```

Scenario C: Chain Overlap (Maximal Grouping)

```
Compute contention: Agents {A1, A2}
Storage contention: Agents {A2, A3}
Dataset contention: Agents {A3, A4}
Memory contention: Agents {A4, A5}
```

```
Transitive chain: A1↔A2↔A3↔A4↔A5
Result: One giant joint contention: {A1, A2, A3, A4, A5} × {compute,
storage, dataset, memory}
Single 5×4=20 variable problem.
```

Scenario D: Universal Resource (Worst Case)

```
Compute contention: All 20 agents request compute
[Plus 9 other resource-specific contentions with various subsets]

Because compute contention overlaps with everything, ALL contentions merge.
Result: One joint contention: 20 agents × 10 resources = 200 variables.
Solve time: ~200-500ms (still within budget).
```

These scenarios demonstrate that grouping behavior adapts to request patterns. When agents have orthogonal resource interests (Scenario A), no grouping occurs and we get parallelism. When agents have overlapping interests (Scenarios B-D), grouping ensures we find all possible cross-resource trades.

3.4 ProportionalFairnessArbitrator: The Mathematical Heart

The ProportionalFairnessArbitrator embodies the platform's core mechanism design, computing allocations that maximize weighted logarithmic utilities subject to resource constraints across potentially multiple resource types simultaneously. This component implements Theorems 1-2: globally Pareto-optimal allocations (Theorem 1, Property 2.3) via polynomial-time convex optimization (Theorem 2). Its correctness is critical—bugs here would violate the mathematical guarantees the entire system depends on.

Core Responsibility:

Given a Contention (which may span multiple resource types and agents due to aggressive grouping) and current GlobalState (containing agent preferences and priority weights), compute an AllocationResult that specifies how much of each resource type each agent receives and how much currency each burns. The allocation must be feasible across all resource types (doesn't exceed available supply for any type, respects agent minimums for all requested resources), globally Pareto-optimal over the joint resource space (on the Pareto frontier considering all resources simultaneously), and computable in polynomial time (no exponential worst-case even for large joint contentions).

Mathematical Foundation for Joint Arbitration:

For a joint contention involving n agents and m resource types, the arbitrator maximizes the Weighted Proportional Fairness objective:

```
maximize: Σᵢ₌₁ⁿ cᵢ · log(Φᵢ(A))


where:
  cᵢ = BaseWeight + CurrencyBurned_i (priority weight for agent i)
  Φᵢ(A) = Σⱼ₌₁ᵐ wᵢⱼ · aᵢⱼ (agent i's utility from joint allocation A)


subject to:
  Σᵢ₌₁ⁿ aᵢⱼ ≤ Qⱼ                    (j = 1, ..., m)   [Resource limits per
type]
  aᵢⱼ ≥ minᵢⱼ                       (∀i, j)           [Minimums per
agent-resource]
  aᵢⱼ ≤ idealᵢⱼ                     (∀i, j)           [Maximums per
agent-resource]
  Σⱼ₌₁ᵐ wᵢⱼ · aᵢⱼ > 0               (i = 1, ..., n)   [Positive utility
per agent]
  aᵢⱼ ∈ ℕ                           (∀i, j)           [Integer
allocations]
```

The key difference from single-resource arbitration is that the optimization now considers the full allocation matrix $A = [a_{ij}]$ where rows are agents and columns are resource types. This enables the mechanism to make tradeoffs across resources: allocate more of resource j_1 to agent i in exchange for allocating more of resource j_2 to agent k, if this improves total weighted logarithmic utility.

Why Joint Optimization Achieves Global Pareto Optimality:

By Theorem 1, any Proportional Fairness allocation is Pareto-optimal within the scope of the optimization. Aggressive grouping sets the scope to include ALL contested resources and ALL agents requesting any contested resource. Therefore, the joint optimization produces an allocation that is Pareto-optimal over the full joint resource space (Property 2.3).

Separate optimizations would each be locally Pareto-optimal but could miss global improvements. The joint optimization considers all dimensions simultaneously, discovering improvements that require cross-resource trades.

Solver Integration: Clarabel with Exponential Cones for Joint Problems:

The logarithmic objective remains most naturally represented using exponential cones, regardless of whether we're optimizing over one resource type or many:

```
maximize: Σᵢ cᵢ · tᵢ


subject to:
```

```
  (tᵢ, Φᵢ(A), 1) ∈ K_exp for all i
  [linear constraints across all resources]

where:
  Φᵢ(A) = Σⱼ wᵢⱼ · aᵢⱼ sums across ALL resource types in the joint contention
  K_exp = {(z, y, 1) : y > 0, y·exp(z/y) ≥ 1}
```

For joint contentions, each agent's utility $\Phi_i(A)$ aggregates their allocation across multiple resource types weighted by their preferences. The exponential cone constraint $(t_i, \Phi_i(A), 1) \in K\_exp$ ensures $t_i = \log(\Phi_i(A))$ at optimality, just as in single-resource case. The solver treats this identically whether Φ_i sums over 1 resource or 10 resources—the mathematics is the same.

Interface Design:

```
public interface ProportionalFairnessArbitrator {
    /**
     * Compute globally Pareto-optimal allocation for contention.
     * Contention may be single-resource or joint (multiple resources).
     *
     * @param contention Resource conflict(s) to resolve (may be joint)
     * @param state Current platform state (for preferences, currency)
     * @return Allocation across all resource types in contention
     * @throws ArbitrationException if optimization fails or infeasible
     */
    AllocationResult arbitrate(Contention contention,
                               ImmutableGlobalState state)
        throws ArbitrationException;
}

public class AllocationResult {
    // Now a matrix: agent × resource type → quantity
    private final Map<AgentId, Map<ResourceType, Long>> allocationMatrix;
    private final Map<AgentId, BigDecimal> currencyBurned;
    private final double objectiveValue;
    private final long computationTimeMs;
    private final SolverMetadata solverMetadata;
    private final boolean isJoint;  // True if multiple resource types
```

```
    // Getters and utilities
    public long getAllocation(AgentId agent, ResourceType resource);
    public Set<ResourceType> getAllocatedResourceTypes();
    public Set<AgentId> getAllocatedAgents();
}
```

The AllocationResult now stores a full allocation matrix rather than a single-resource map. For single-resource contentions, the matrix has one column. For joint contentions, it has multiple columns, one per contested resource type.

Arbitration Process for Joint Contentions:

The arbitration process handles both single-resource and joint contentions uniformly:

1. **Extract preferences:** Query state for each involved agent's PreferenceFunction
2. **Query priority budgets:** Call each agent's `decidePriorityBudget()` to learn how much currency they want to burn
3. **Validate currency:** Check agents have sufficient balances to burn requested amounts
4. **Compute priority weights:** c_i = BaseWeight + CurrencyBurned_i for each agent
5. **Formulate joint problem:** Convert to exponential cone program for Clarabel spanning all resource types
6. **Solve:** Invoke Clarabel, get continuous-valued solution across full allocation matrix
7. **Round to integers:** Apply rounding with feasibility preservation (see Section 8.2)
8. **Validate solution:** Check all constraints satisfied across all resource types, compute objective value
9. **Package result:** Create AllocationResult with full allocation matrix, currency, metadata
10. **Return:** Hand result to ResourceManager for atomic enforcement

Step 5 (formulate joint problem) is more complex than single-resource case because we must correctly set up the multi-dimensional optimization. The utility for each agent sums across all m resource types in the contention, weighted by that agent's preference weights for each type.

Priority Weight Calculation:

Priority weights are computed by the PriorityWeightCalculator (detailed in Section 3.6), which the arbitrator calls as a dependency. The arbitrator doesn't implement Priority Economy logic itself—it just uses priority weights in the objective function. This separation of concerns keeps the arbitrator focused on optimization mechanics while the calculator handles economic dynamics.

The priority weight formula remains simple regardless of whether arbitration is single-resource or joint:

```
c_i = BaseWeight + CurrencyBurned_i

where BaseWeight = 10.0 (ensures participation even with zero currency)
```

The BaseWeight is critical for starvation protection. Even agents with zero currency (either because they're new, or because they've depleted their balance) get priority weight 10.0, ensuring they participate in allocations. Agents who burn currency increase their weight above baseline, getting preferentially allocated resources across all contested types, but the logarithmic objective prevents them from monopolizing any resource (as any agent's utility approaches zero, reallocating to them produces infinite marginal gain in the objective).

This creates a continuous spectrum of outcomes rather than discrete tiers. An agent burning 0 currency has weight 10, receiving baseline allocation. An agent burning 50 currency has weight 60 (6x baseline), receiving substantially more but not a monopoly. An agent burning 500 currency has weight 510 (51x baseline), receiving dominant allocation but still constrained by logarithmic barrier from excluding others. The spectrum is smooth—small increases in spending produce small increases in allocation, with diminishing returns.

Performance Characteristics:

For Milestone 1's target scale, empirical performance with Clarabel on joint contentions:

Small joint contentions (n=5, m=3):
- Variables: 15
- Solve time: 20-50ms
- Iterations: 15-25

Moderate joint contentions (n=10, m=5):
- Variables: 50
- Solve time: 50-150ms
- Iterations: 20-35

Large joint contentions (n=20, m=10):
- Variables: 200
- Solve time: 200-500ms
- Iterations: 25-45

Worst case (n=20, m=10 with poor conditioning):
- Variables: 200
- Solve time: 500-800ms (99th percentile)
- Iterations: 40-60

These characteristics are well within the 1-second arbitration budget even for worst-case joint contentions at Milestone 1 scale. The polynomial scaling O((nm)^2.5) means doubling problem size increases solve time by ~5.7x, which is predictable and manageable.

Error Handling and Robustness:

The arbitrator must handle several error conditions gracefully, with additional considerations for joint contentions:

Infeasible minimums across multiple resources: If constraints are unsatisfiable across the joint resource space (agents' minimums cannot all be satisfied simultaneously across all

contested resource types), the problem is infeasible. The arbitrator detects this through a preliminary feasibility check or through solver returning INFEASIBLE status. For joint contentions, infeasibility might arise from complex interactions: Agent A's minimum for compute plus Agent B's minimum for storage might individually be satisfiable, but satisfying both simultaneously while respecting Agent C's minimums might be impossible.

Detection: Attempt to solve feasibility problem (find any allocation satisfying all minimums and resource limits, ignoring optimality). If no solution exists, signal InfeasibleJointContentionException with details about which constraints conflict.

Solver failure: Clarabel might fail to converge (rare but possible for pathological joint problem instances with poor conditioning across multiple resource types). The arbitrator catches solver exceptions, logs details including problem size and structure, and throws ArbitrationException. The platform can retry with perturbed parameters or skip this arbitration cycle.

Numerical instability: Floating-point rounding errors might cause slight constraint violations in the joint allocation matrix. The arbitrator applies epsilon-tolerance checks (constraint violated if violation > $\varepsilon = 10^{-6}$), ignoring tiny numerical errors while catching real violations. For joint problems with 200 variables, accumulated rounding error could be higher than single-resource problems, requiring careful tolerance tuning.

Partial solver success: If solver times out but returns partial solution (has improved but not fully converged), the arbitrator can either accept the partial solution (near-optimal but not guaranteed Pareto-optimal) or reject and retry. For Milestone 1, we reject partial solutions to maintain optimality guarantee. Phase 2 might accept with explicit approximation bounds.

These error handlers ensure the arbitrator fails safely—it either returns a valid AllocationResult (satisfying all constraints, provably optimal) or throws an exception explaining why it couldn't. It never returns invalid allocations (violating constraints across any resource type) or silent failures (allocating incorrectly without signaling error).

3.5 PriorityWeightCalculator: Priority Economy Management

The PriorityWeightCalculator implements the Priority Economy's closed-loop dynamics: agents earn currency by releasing resources back to the pool, spend currency by burning it to increase priority weights, and the platform tracks balances to prevent overspending. This component bridges the arbitration mechanism (which uses priority weights in optimization) and the resource economy (which generates and destroys currency).

Core Responsibilities:

Priority weight calculation: Given an agent's currency commitment (how much they want to burn), compute their priority weight c_i = BaseWeight + CurrencyBurned_i. Validate that the agent has sufficient balance to burn the requested amount.

Earning calculation: When an agent releases resources, compute how much currency they earn using the formula: Income = Quantity × TimeRemaining × CurrentDemandMultiplier. This formula incentivizes early release (TimeRemaining is higher if released before lease expires) and releasing scarce resources (DemandMultiplier is higher when demand exceeds supply). For

joint allocations involving multiple resource types, earnings are calculated per resource type and summed.

Demand multiplier tracking: For each resource type, continuously update CurrentDemandMultiplier = TotalDemand / AvailableSupply. This provides a dynamic scarcity signal—resources in high demand have high multipliers, making their release more lucrative.

Balance management: Track each agent's currency balance, enforcing non-negativity with small bounded debt (minimum balance = -100 allows temporary overspending). Currency is created when resources are released, destroyed when priority weights are calculated, never transferred between agents.

Interface Design:

```
public interface PriorityWeightCalculator {
    /**
     * Calculate priority weights from currency commitments.
     * Validates agents have sufficient balances.
     *
     * @param commitments Map of agent ID to currency burn amounts
     * @param state Current state (for balance checking)
     * @return Map of agent ID to priority weight (c_i = BaseWeight + burned)
     * @throws InsufficientCurrencyException if any agent lacks balance
     */
    Map<AgentId, Double> calculatePriorityWeights(
        Map<AgentId, BigDecimal> commitments,
        ImmutableGlobalState state
    ) throws InsufficientCurrencyException;

    /**
     * Calculate earnings for releasing resources.
     * Handles both single-resource and multi-resource releases.
     *
     * @param resources Bundle of resources being released
     * @param leaseExpirations When leases would naturally expire (per resource type)
     * @param currentTime When release is happening (for TimeRemaining)
     * @param state Current state (for demand multipliers)
     * @return Total currency earned (summed across all resource types)
     */
```

```
    BigDecimal calculateReleaseEarnings(
        ResourceBundle resources,
        Map<ResourceType, Instant> leaseExpirations,
        Instant currentTime,
        ImmutableGlobalState state
    );

    /**
     * Calculate current demand multiplier for resource type.
     * Multiplier = TotalDemand / AvailableSupply.
     *
     * @param resourceType Resource to compute multiplier for
     * @param state Current state (for pending requests and pool)
     * @return Multiplier (typically 0.5 to 3.0, higher = more scarce)
     */
    BigDecimal calculateDemandMultiplier(
        ResourceType resourceType,
        ImmutableGlobalState state
    );
}
```

The interface makes explicit the calculator's three main responsibilities: weights (for arbitration), earnings (for releases), and multipliers (for scarcity signaling). Each responsibility has a dedicated method with clear inputs and outputs, enabling independent testing and validation. The calculator handles both single-resource and joint scenarios uniformly—the earning formula applies per resource type and sums across all types in the release bundle.

Priority Weight Calculation:

Priority weight calculation is the simplest responsibility:

```
For each agent i with commitment c_i:
  1. Validate: balance_i ≥ c_i (or balance_i ≥ -100 if allowing debt)
  2. Calculate: weight_i = BaseWeight + c_i
  3. Return: map {agent_i → weight_i}
```

The BaseWeight (10.0) ensures that even agents with zero currency get meaningful priority. This is the mathematical foundation of starvation protection—the logarithmic objective means an

agent with weight 10.0 still gets allocation sufficient to keep their utility positive, even if competing against agents with weight 510.0 (who burned 500 currency).

The validation step is critical for preventing overspending bugs. If an agent claims they'll burn 100 currency but only has 50, we catch this before arbitration. The arbitrator receives only valid commitments, simplifying its logic. The bounded debt (-100 minimum balance) provides flexibility for temporary overspending (agent burns 110 currency when they have 100, going -10 in debt), which they must earn back through future releases.

Earning Calculation:

Earning calculation implements the Priority Economy's incentive structure, now with explicit support for multi-resource releases:

```
For each resource type j being released:
  quantity_j = amount released
  leaseExpiry_j = when lease would naturally expire
  currentTime = when release happens
  TimeRemaining_j = leaseExpiry_j - currentTime (in timesteps or seconds)
  DemandMultiplier_j = calculateDemandMultiplier(j, state)

  earnings_j = quantity_j × TimeRemaining_j × DemandMultiplier_j

Total earnings = Σ_j earnings_j
```

This formula has several nice properties:

Early release incentive: TimeRemaining is higher if agent releases resources before lease expires. An agent with a 10-timestep lease who releases after 2 timesteps earns 8× more per unit than if they release at expiration. This incentivizes returning unused capacity promptly rather than hoarding.

Scarcity premium: DemandMultiplier is higher when resources are scarce. An agent releasing compute when DemandMultiplier=2.0 (demand is 2× supply) earns double what they'd earn when DemandMultiplier=1.0. This incentivizes releasing high-demand resources, naturally equilibrating supply.

Quantity proportionality: Earnings scale with quantity released. Releasing 100 units earns 10× releasing 10 units (other factors equal). This creates pressure to acquire resources only when needed—holding excess resources prevents others from using them, and the opportunity cost (foregone earnings from release) is high.

Multi-resource independence: For agents releasing multiple resource types simultaneously (common after joint allocations), each resource type's earnings are computed independently and summed. This simplification (no cross-resource effects) keeps the formula tractable. Phase

2 might add complementarity bonuses (extra earnings for releasing complementary resources together), but Milestone 1's independence is sufficient.

Demand Multiplier Calculation:

Demand multipliers provide real-time scarcity signals, calculated independently per resource type:

```
For resource type j:
  AvailableSupply_j = current pool quantity
  TotalDemand_j = Σ_i (agent i's current ideal request for j)

  If AvailableSupply_j > 0:
    DemandMultiplier_j = TotalDemand_j / AvailableSupply_j
  Else:
    DemandMultiplier_j = FIXED_HIGH_VALUE (e.g., 10.0)
```

Typical demand multipliers range from 0.5 (low demand, supply exceeds demand) to 3.0 (high demand, demand is 3× supply). Extreme values (multiplier > 5.0) are possible during resource exhaustion, signaling critical scarcity.

The edge case handling (supply = 0) uses a fixed high value rather than infinity because we want well-defined earnings even when pool is temporarily exhausted. The high value (10.0) provides strong incentive to release resources into an empty pool without creating numerical instability from infinite multipliers.

Currency Balance Tracking:

The calculator doesn't maintain balances itself—balances live in the ImmutableGlobalState's ResourceLedger. But the calculator is responsible for computing how balances should change:

Burning (in calculatePriorityWeights): When arbitration occurs, the calculator computes weights and returns how much currency each agent burned. The ResourceManager (via TransactionManager) then calls `ledger.burnCurrency(agentId, amount)`, which decrements the balance by the burned amount (destroying currency permanently).

Minting (in calculateReleaseEarnings): When resources are released, the calculator computes earnings per resource type and sums them. The ResourceManager calls `ledger.mintCurrency(agentId, earnings)`, which increments the balance by the earned amount (creating new currency).

This separation between calculation and mutation keeps the calculator stateless and pure. It doesn't have side effects; it just computes numbers. The ResourceManager handles actual state mutations, ensuring they're part of atomic state transitions (particularly important for joint allocations where multiple resource types are released together).

Equilibrium Dynamics:

Over time, the Priority Economy reaches equilibrium where agents' earning rates balance their spending rates. The equilibrium currency balance $\bar{C}_i$ for agent i depends on their structural behavior:

Frequent releasers (agents who acquire, use briefly, release) earn often, maintaining high balances. They can afford to burn currency regularly for priority when needed. For agents involved in joint allocations, frequent release of multiple resource types accelerates earning.

Infrequent releasers (agents who hold resources for long periods) earn rarely, maintaining low balances. They must spend currency cautiously, saving it for truly urgent needs.

Aggressive burners (agents who frequently bid high for priority) deplete balances faster than they earn, eventually reaching low equilibrium. They get good allocations when they have currency but suffer when depleted.

Conservative burners (agents who burn minimally) accumulate currency over time, reaching high equilibrium. They forego priority wins but maintain financial stability.

The system self-regulates: agents who overspend run low on currency, forcing them to reduce spending, which allows their balances to recover through earning. Agents who underspend accumulate currency, eventually hitting spending opportunities where burning makes sense. This self-regulation is what makes the Priority Economy stable without requiring platform intervention to balance budgets.

3.6 ResourceManager: Ownership and Transfers with Atomic Joint Enforcement

The ResourceManager is responsible for enforcing allocations computed by the arbitrator, tracking resource leases across multiple resource types, handling releases, and maintaining resource conservation invariants. For joint allocations involving multiple resource types, the ResourceManager ensures atomic enforcement—either all resources for all agents are allocated together, or none are (Property 6.3).

Core Responsibilities:

Joint allocation enforcement: When arbitrator computes joint allocation (spanning multiple resource types and agents), ResourceManager coordinates with TransactionManager to apply all changes atomically: transfers resources across all types, burns currency from all agents, records lease expirations for all allocated resources, all in one atomic transaction.

Release processing: When agents release resources (potentially multiple types from joint allocations), ResourceManager transfers resources from agent back to pool, calculates earnings summed across all released types, mints currency into agent balance.

Lease tracking: Maintains map of when each agent's resource leases expire (per resource type), enables automatic reclamation when leases expire, provides data for TimeRemaining calculation in earnings formula.

Conservation enforcement: Before every state transition involving resources, validates that resource conservation holds for ALL resource types (total quantity unchanged per type), rejects transitions that would violate conservation for any type.

Interface:

```
public interface ResourceManager {
    /**
     * Enforce allocation by updating ledger atomically.
     * For joint allocations, transfers ALL resource types atomically.
     * Transfers resources from pool to agents, burns currency.
     * Returns new state with updated ledger and balances.
     *
     * @param result Allocation to enforce (may span multiple resource
types)
     * @param currentState Current state to transition from
     * @return New state with allocations applied atomically
     * @throws InsufficientResourcesException if allocation exceeds
availability
     * @throws TransactionException if atomic commit fails
     */
    GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState
    ) throws InsufficientResourcesException, TransactionException;

    /**
     * Process resource release (potentially multiple types).
     * Transfers resources back to pool, calculates and mints earnings.
     *
     * @param agentId Agent releasing resources
     * @param resources Bundle being released (may contain multiple types)
     * @param currentState Current state
     * @return New state with releases applied and earnings minted
     */
    GlobalState releaseResources(
        AgentId agentId,
        ResourceBundle resources,
```

```
        GlobalState currentState
    );

    /**
     * Check if allocation is feasible given current pool state.
     * Validates across ALL resource types in allocation.
     * Used for pre-validation before enforcement.
     */
    boolean isAllocationFeasible(
        AllocationResult result,
        GlobalState state
    );

    /**
     * Get lease expiration time for agent's resource holdings.
     * Returns per-resource-type expiration map.
     */
    Map<ResourceType, Instant> getLeaseExpirations(
        AgentId agentId,
        GlobalState state
    );
}
```

Atomic Enforcement Process for Joint Allocations:

When enforceAllocation() is called with a joint AllocationResult:

1. **Validate feasibility across all types:** Check that pool has sufficient resources for ALL contested resource types (sanity check—arbitrator should ensure this, but defense-in-depth)
2. **Validate currency:** Check that agents have sufficient balances to burn requested amounts
3. **Begin transaction:** Call `transactionManager.begin()`
4. **Within transaction scope:**
   - **Transfer resources:** For each agent and each resource type in the allocation matrix, `ledger.transfer(POOL_ID, agentId, resourceType, quantity)`
   - **Burn currency:** For each agent, `ledger.burnCurrency(agentId, amount)`
   - **Record lease expirations:** For each (agent, resource type) pair allocated, `ledger.setLeaseExpiration(agentId, resourceType, now + leaseDuration)`
5. **Pre-commit validation:** SafetyMonitor checks all invariants on hypothetical new state
6. **If validation passes:** `transactionManager.commit()` makes state changes visible atomically
7. **If validation fails:** `transactionManager.rollback()` discards all changes, state unchanged
8. **Return result:** New state (if committed) or exception (if rolled back)

The key architectural property is that steps 4-6 form an atomic transaction (Property 6.3). Either ALL resource types are allocated to ALL agents successfully, or NONE are. No partial allocations where some agents get compute but not storage, or where resources transfer but currency doesn't burn.

Why Atomicity Matters for Joint Allocations:

Without atomicity, failure in middle of enforcement could leave the system in inconsistent state:

Bad scenario without atomicity:

```
Joint allocation: {A₁, A₂} × {compute, storage}
1. Allocate compute to A₁, A₂ ✓
2. Allocate storage to A₁ ✓
3. Allocate storage to A₂ ✗ (error: insufficient pool storage)
4. Burn currency from A₁, A₂ ✓

Result: A₁ has compute+storage (good), A₂ has compute but not storage
(broken),
        both burned currency (can't undo), system inconsistent.
```

With atomicity, step 3's failure triggers rollback of steps 1-2. Both agents retain their previous state, no currency burned, system remains consistent. They can retry next arbitration cycle.

Release Process for Multi-Resource Bundles:

When releaseResources() is called with a bundle containing multiple resource types:

1. **Validate ownership:** Check that agent actually holds ALL resources in the bundle
2. **Calculate TimeRemaining per type:** For each resource type, `timeRemaining_j = leaseExpiration_j - now`
3. **Get demand multipliers:** Query for current multipliers for ALL types in bundle
4. **Calculate earnings per type:** `earnings_j = quantity_j × timeRemaining_j × multiplier_j`
5. **Sum total earnings:** `totalEarnings = Σ_j earnings_j`
6. **Create new ledger:** Copy current ledger using structural sharing
7. **Transfer resources:** For each type, `ledger.transfer(agentId, POOL_ID, resourceType, quantity)`
8. **Mint currency:** `ledger.mintCurrency(agentId, totalEarnings)` (creates new currency)
9. **Clear lease expirations:** Remove lease records for all released resources
10. **Create new state:** Build new GlobalState with updated ledger
11. **Return new state:** Hand to platform for commit

The multi-resource handling is straightforward because earnings are independent per resource type. An agent releasing {compute:50, storage:30} earns `50×timeRemaining_compute×multiplier_compute + 30×timeRemaining_storage×multiplier_storage`. No complex interactions between resource types—just sum the earnings.

3.7 TransactionManager: Atomic Operations for Joint Allocations

The TransactionManager is introduced to provide explicit transaction semantics for joint allocations. While Milestone 1's implementation leverages immutability for implicit transaction behavior (build complete new state, commit atomically), the TransactionManager interface establishes a clear abstraction for atomic multi-step operations that would also become essential in any future distributed deployment.

Purpose and Scope:

For Milestone 1-2, the TransactionManager is primarily an architectural placeholder that makes transaction semantics explicit. The actual implementation is simple because immutable state provides natural atomicity. For a later scope with a distributed platform across multiple nodes, the TransactionManager becomes critical—coordinating atomic commits across nodes via two-phase commit or similar protocols.

Interface Design:

```
public interface TransactionManager {
    /**
     * Begin a new transaction.
     * Allocates transaction ID, prepares for multi-step atomic operation.
     *
     * @return Transaction handle for subsequent operations
     */
    Transaction begin();

    /**
     * Execute operation within transaction scope.
     * Operation builds modified state but doesn't commit yet.
     *
     * @param txn Transaction handle
     * @param operation State-modifying operation
     * @return Modified state (uncommitted)
     */
    GlobalState execute(Transaction txn, StateOperation operation)
        throws TransactionException;

    /**
     * Commit transaction, making all changes visible atomically.
```

```
     * Validates via SafetyMonitor before committing.
     *
     * @param txn Transaction to commit
     * @param finalState State to commit (result of all operations)
     * @return Committed state with new version number
     * @throws TransactionException if validation fails or commit
impossible
     */
    GlobalState commit(Transaction txn, GlobalState finalState)
        throws TransactionException;

    /**
     * Rollback transaction, discarding all changes.
     * Returns to state before transaction began.
     *
     * @param txn Transaction to rollback
     * @param originalState State to return to
     */
    void rollback(Transaction txn, GlobalState originalState);
}

public interface Transaction {
    UUID getTransactionId();
    Instant getStartTime();
    TransactionStatus getStatus();  // ACTIVE, COMMITTED, ROLLED_BACK
}

@FunctionalInterface
public interface StateOperation {
    /**
     * Apply operation to state, returning modified state.
     * Should be pure function (no side effects).
     */
    GlobalState apply(GlobalState currentState) throws Exception;
}
```

Milestone 1-2 Implementation Pattern:

For single-JVM deployment with immutable state, the implementation is straightforward:

```
public class SimpleTransactionManager implements TransactionManager {
    private final SafetyMonitor safetyMonitor;

    @Override
    public Transaction begin() {
        return new SimpleTransaction(UUID.randomUUID());
    }

    @Override
    public GlobalState execute(Transaction txn, StateOperation operation)
            throws TransactionException {
        try {
            // Simply execute operation (builds new state via immutability)
            return operation.apply(currentState);
        } catch (Exception e) {
            throw new TransactionException("Operation failed", e);
        }
    }

    @Override
    public GlobalState commit(Transaction txn, GlobalState finalState)
            throws TransactionException {

        // Validate safety before commit
        if (!safetyMonitor.validateState(finalState).isValid()) {
            throw new TransactionException(
                "Safety validation failed, transaction rolled back"
            );
        }

        // Commit is just returning the final state
        // Caller will update currentState reference atomically
        txn.setStatus(TransactionStatus.COMMITTED);
        return finalState;
    }
```

```
    @Override
    public void rollback(Transaction txn, GlobalState originalState) {
        // Rollback is trivial: just don't use finalState
        // Caller retains originalState reference
        txn.setStatus(TransactionStatus.ROLLED_BACK);
    }
}
```

This implementation might seem trivial (it barely does anything!), but it serves important architectural purposes:

Explicit transaction boundaries: Code that performs multi-step operations must explicitly begin/commit transactions, making transaction scope visible in the codebase.

Extension point for distribution: A later scope's distributed implementation would replace this with sophisticated two-phase commit across nodes, but the interface would remain the same.

Centralized validation: All commits go through validation, ensuring consistency regardless of who initiated the transaction.

Auditability: Transaction IDs enable tracing which operations were part of which transactions in event logs.

Usage Pattern in ResourceManager:

```
public GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState) {

    // Begin transaction
    Transaction txn = transactionManager.begin();

    try {
        // Execute all resource transfers within transaction
        GlobalState afterTransfers = transactionManager.execute(txn, state
-> {
            GlobalState s = state;

            // Transfer each resource type for each agent
            for (AgentId agent : result.getAllocatedAgents()) {
```

```
                for (ResourceType type :
result.getAllocatedResourceTypes()) {
                    long quantity = result.getAllocation(agent, type);
                    if (quantity > 0) {
                        s = s.transferResource(POOL_ID, agent, type,
quantity);
                    }
                }
            }

            return s;
        });

        // Execute currency burning within same transaction
        GlobalState afterBurning = transactionManager.execute(txn, state ->
{
            GlobalState s = state;

            for (Map.Entry<AgentId, BigDecimal> entry :
                    result.getCurrencyBurned().entrySet()) {
                s = s.burnCurrency(entry.getKey(), entry.getValue());
            }

            return s;
        });

        // Execute lease recording within same transaction
        GlobalState afterLeases = transactionManager.execute(txn, state ->
{
            GlobalState s = state;
            Instant now = Instant.now();

            for (AgentId agent : result.getAllocatedAgents()) {
                for (ResourceType type :
result.getAllocatedResourceTypes()) {
                    if (result.getAllocation(agent, type) > 0) {
                        Duration leaseDuration = getLeaseDuration(type);
                        s = s.setLeaseExpiration(agent, type,
                                                 now.plus(leaseDuration));
```

```
                    }
                }
            }

            return s;
        });

        // Commit transaction (validates and makes visible)
        GlobalState committed = transactionManager.commit(txn,
afterLeases);
        return committed;

    } catch (Exception e) {
        // Any failure triggers rollback
        transactionManager.rollback(txn, currentState);
        throw new ResourceManagementException(
            "Failed to enforce joint allocation", e
        );
    }
}
```

This pattern makes explicit the multiple steps involved in enforcement and ensures they're atomic. For Milestone 1, the "transaction" is mostly documentation (immutability already provides atomicity), but the pattern establishes good practices for later scopes.

---

4. Data Models and State Management

The platform's data models define what information exists, how it's structured, and how it evolves over time. This section specifies the core data types (GlobalState, ResourceLedger, Contention, AgentConfiguration) and explains the immutable state management approach that provides thread-safety, reproducibility, and auditability.

4.1 ImmutableGlobalState: The Root of All State

ImmutableGlobalState is the authoritative record of platform state at a specific version. It contains everything needed to understand the system at a point in time: resource ownership, agent states, active commitments, and event history. The "immutable" prefix is not

decorative—instances are truly immutable once created, using defensive copying and persistent data structures to prevent modifications.

Structure:

```
public final class ImmutableGlobalState {
    private final long version;  // monotonically increasing
    private final Instant timestamp;
    private final ImmutableResourceLedger resourceLedger;
    private final ImmutableMap<AgentId, ImmutableAgentState> agentStates;
    private final ImmutableCommitmentGraph commitments;
    private final ImmutableList<Event> eventHistory;

    // Private constructor enforces factory pattern
    private ImmutableGlobalState(...) { ... }

    // Factory method for initial state
    public static ImmutableGlobalState createInitial(
        ResourcePool pool,
        Collection<AgentConfiguration> agents
    );

    // State transition method (returns new instance)
    public ImmutableGlobalState applyEvent(Event event);

    // Accessors (all return immutable views)
    public long getVersion();
    public ImmutableResourceLedger getResourceLedger();
    public ImmutableAgentState getAgentState(AgentId id);
    public ImmutableList<Event> getEventHistory();
    public Set<ResourceType> getResourceTypes();
}
```

Version Tracking:

The version number is a monotonically increasing counter starting at 0 (initial state). Every state transition increments the version by 1. This creates a total ordering of states: if version_A < version_B, state A precedes state B in time. Version numbers enable optimistic concurrency

(check that we're transitioning from the expected version) and temporal queries (what was the state at version V?).

Structural Sharing:

Creating a new state version for every event seems expensive—if state is 50MB and we process 100 events/second, that's 5GB/second of allocation. But structural sharing makes this practical. When we update one agent's state (changing their resource allocation across multiple types after joint arbitration), we don't copy the entire GlobalState. Instead:

1. **Copy the path** from root to changed nodes (agent state for affected agents, ledger entries for modified resources)
2. **Share everything else** (other agents' states, unmodified ledger portions, event history)
3. **Link** the new version to unchanged portions

This reduces copying from O(state_size) to O(log(state_size)) using tree-structured persistent data structures. For our use case (state size ~50MB, joint updates affect ~2-10 agents × 2-10 resource types per event), structural sharing keeps overhead at ~1-5MB per state transition, acceptable.

We use Java libraries like PCollections or Vavr that provide persistent implementations of Map, List, Set. These libraries handle structural sharing automatically—we just call `map.put(key, value)` and receive a new map that shares structure with the old one.

Event History:

The eventHistory field maintains a complete log of all events that have been applied. This provides an audit trail (what decisions were made?) and enables event replay (apply same events to initial state, reproduce current state). For debugging, event replay is invaluable—when something unexpected happens, we can replay events step by step, inspecting state after each transition.

For long-running platforms, event history might grow large (10,000 events × 1KB each = 10MB). Phase 2 can add history pruning (keep only recent N events) or snapshotting (periodically save full state, discard events before snapshot). Milestone 1 keeps full history for maximum debugging capability.

State Transition Semantics:

State transitions are implemented as pure functions:

```
public ImmutableGlobalState applyEvent(Event event) {
    return switch(event.getType()) {
        case RESOURCE_REQUEST -> handleResourceRequest(event);
        case CONTENTION_DETECTED -> handleContentionDetected(event);
        case ARBITRATION_COMPLETE -> handleArbitrationComplete(event);
        case RESOURCE_RELEASE -> handleResourceRelease(event);
        case ALLOCATION_ENFORCED -> handleAllocationEnforced(event);
```

```
    };
}
```

Each handler method:
1. **Validates** event is applicable to current state
2. **Computes** what should change (which agents, which resources, which balances)
3. **Creates** new immutable versions of changed components
4. **Constructs** new GlobalState with changed components, sharing unchanged ones
5. **Returns** new state (old state remains unchanged)

This functional approach ($state_1$ + event → $state_2$, no mutation of $state_1$) provides several guarantees:

Determinism: Applying the same event to the same state always produces the same result. No hidden dependencies on global state, thread timing, or random values.

Thread-safety: Multiple threads can read $state_1$ while another thread computes $state_2$. No locking needed for reads.

Rollback: To revert to previous state, just keep a reference to it. No complex undo logic needed. Essential for transaction rollback in joint arbitrations.

Testing: Test state transitions by asserting `$state_2$ = $state_1$.applyEvent(event)`. Pure functions are trivial to test.

4.2 ResourceLedger: Tracking Ownership and Currency

The ResourceLedger is a subcomponent of GlobalState that tracks resource ownership (who has which resources) and currency balances (how much currency each agent holds). It enforces conservation invariants and provides query methods for allocation decisions. For joint allocations, the ledger handles updates across multiple resource types atomically through the immutable state model.

Structure:

```
public final class ImmutableResourceLedger {
    // Resource ownership: (AgentId, ResourceType) -> Quantity
    private final ImmutableMap<AgentId, ImmutableMap<ResourceType, Long>>
        resourceHoldings;

    // Currency balances: AgentId -> Balance
    private final ImmutableMap<AgentId, BigDecimal> currencyBalances;

    // Lease tracking: (AgentId, ResourceType) -> Expiration time
```

```
    private final ImmutableMap<AgentId, ImmutableMap<ResourceType,
Instant>>
        leaseExpirations;

    // Currency accounting metadata
    private final BigDecimal cumulativeMinting;
    private final BigDecimal cumulativeBurning;

    // Private constructor
    private ImmutableResourceLedger(...) { ... }

    // Factory for initial ledger
    public static ImmutableResourceLedger createInitial(
        ResourcePool pool,
        BigDecimal initialCurrencyPerAgent
    );

    // Query methods
    public long getResourceQuantity(AgentId agent, ResourceType type);
    public BigDecimal getCurrencyBalance(AgentId agent);
    public long getPoolQuantity(ResourceType type);
    public Instant getLeaseExpiration(AgentId agent, ResourceType type);

    // State transition methods (return new ledger)
    public ImmutableResourceLedger transfer(
        AgentId from, AgentId to, ResourceType type, long quantity
    );

    public ImmutableResourceLedger burnCurrency(
        AgentId agent, BigDecimal amount
    );

    public ImmutableResourceLedger mintCurrency(
        AgentId agent, BigDecimal amount
    );

    public ImmutableResourceLedger setLeaseExpiration(
        AgentId agent, ResourceType type, Instant expiry
```

```
    );

    // Multi-resource atomic operations (for joint allocations)
    public ImmutableResourceLedger transferBundle(
        AgentId from, AgentId to, ResourceBundle bundle
    );

    public ImmutableResourceLedger setLeaseExpirations(
        AgentId agent, Map<ResourceType, Instant> expirations
    );
}
```

Multi-Resource Transfer Support:

The `transferBundle()` method applies multiple resource transfers atomically. Internally, it's implemented as:

```
public ImmutableResourceLedger transferBundle(
        AgentId from, AgentId to, ResourceBundle bundle) {

    ImmutableResourceLedger ledger = this;

    // Apply all transfers, building new ledger incrementally
    for (Map.Entry<ResourceType, Long> entry : bundle.entrySet()) {
        ledger = ledger.transfer(from, to, entry.getKey(),
entry.getValue());
    }

    // Return final ledger with all transfers applied
    return ledger;
}
```

Because each `transfer()` returns a new immutable ledger, this composition builds the final result incrementally while sharing structure. The final ledger reflects all transfers applied, and either we use this final ledger (commit transaction) or we discard it (rollback), ensuring atomicity.

Resource Ownership Model:

Resources are owned by agents or by the ResourcePool (a special pseudo-agent representing unallocated resources). Ownership is tracked with nested maps: outer map keyed by AgentId, inner map keyed by ResourceType, values are quantities. This structure enables efficient queries:

- **Agent holdings:** `ledger.getResourceQuantity(agent, type)` is O(1) lookup
- **Pool availability:** `ledger.getPoolQuantity(type)` is O(1) lookup
- **Total quantity:** Sum across all agents plus pool (for conservation check)

When resources are allocated (transferred from pool to agent) or released (transferred from agent to pool), we create a new ledger with updated quantities. The transfer is atomic from the ledger's perspective—either both sides update or neither does (enforced by immutability: we build complete new ledger before returning it).

Currency Tracking:

Currency balances are tracked per agent as BigDecimal (arbitrary-precision decimal) to avoid floating-point rounding errors. Currency can be burned (decreased by burning for priority) or minted (increased by earning from releases). The ledger doesn't implement earning/burning logic (that's in PriorityWeightCalculator); it just provides primitives for incrementing/decrementing balances.

Currency balances have a floor: `balance ≥ -100` (bounded debt). This allows temporary overspending (agent burns 110 when they have 100) which must be repaid through future earnings. The debt limit prevents unbounded negative balances that could destabilize the economy. For Milestone 1, -100 is a conservative limit; Phase 2 might adjust based on empirical observations.

Lease Expiration Tracking:

When resources are allocated to an agent, the platform records when the lease expires for each resource type. For joint allocations, multiple resource types have leases set simultaneously. The leaseExpirations map tracks these deadlines per (agent, resource type) pair, enabling:

Automatic reclamation: When lease expires, platform can automatically release resources (if agent hasn't released voluntarily)

Earning calculation: TimeRemaining = leaseExpiry - currentTime determines earnings when agent releases early

Monitoring: Platform can warn agents about upcoming expirations or extend leases on request

For Milestone 1, leases are simple: fixed duration from allocation time (may vary by resource type: compute gets 20 timesteps, lab_time gets 5 timesteps). Phase 2 might add complex lease types (renewable leases, conditional extensions, dynamic durations based on contention).

Conservation Invariant:

The ResourceLedger enforces resource conservation as an architectural invariant, now with explicit support for validating across all resource types:

```
public boolean checkConservation(
        Map<ResourceType, Long> initialQuantities) {

    for (Map.Entry<ResourceType, Long> entry :
initialQuantities.entrySet()) {
        ResourceType type = entry.getKey();
        long expectedTotal = entry.getValue();

        // Sum across all agents
        long totalHeld = resourceHoldings.values().stream()
            .mapToLong(agentMap -> agentMap.getOrDefault(type, 0L))
            .sum();

        // Add pool quantity
        totalHeld += getPoolQuantity(type);

        if (totalHeld != expectedTotal) {
            logger.error("Conservation violated for {}: total={},
expected={}",
                        type, totalHeld, expectedTotal);
            return false;
        }
    }

    return true;
}
```

This check is called by SafetyMonitor before every state transition that touches resources. If conservation is violated for ANY resource type (sum of holdings ≠ initial quantity), the transition is rejected. This prevents accounting bugs that could create or destroy resources inadvertently, which is particularly important for joint allocations where many resource types are updated simultaneously.

Currency conservation is more nuanced: currency is intentionally created (minting when resources released) and destroyed (burning for priority). The "conservation" for currency is: `$\Sigma_i$ balance_i + cumulativeBurning = cumulativeMinting + InitialEndowments`. Total currency in existence equals initial endowments plus all minting minus all burning. This can be verified using the cumulative counters tracked in ledger metadata.

4.3 AgentConfiguration and AgentState

Agents have two related but distinct representations: **AgentConfiguration** (the hybrid artifact submitted to platform, defining preferences and behavior) and **AgentState** (the runtime state tracking resources, goals, and progress). The configuration is static (immutable after submission), while state evolves through interactions.

AgentConfiguration:

```
public class AgentConfiguration {
    private final AgentId agentId;
    private final String version;

    // Declarative components (data)
    private final PreferenceFunction preferences;
    private final List<Goal> goals;
    private final ResourceBundle initialResources;
    private final Set<ResourceType> resourceInterests;

    // Executable component (code)
    private final Class<? extends AgentBehavior> behaviorClass;

    // Constructor, getters

    // Factory method: load from JSON + compile behavior class
    public static AgentConfiguration loadFrom(
        String jsonConfig,
        ClassLoader behaviorClassLoader
    );
}
```

Configurations are created through a factory pattern that combines JSON parsing (for declarative components) with class loading (for behavioral component):

1. **Parse JSON** to extract agent ID, preferences, goals, initial resources
2. **Validate preference weights** sum to 1.0, are positive, reference valid resource types
3. **Validate goals** reference valid resources, thresholds are achievable
4. **Load behavior class** from specified fully-qualified name
5. **Instantiate configuration** object with all components
6. **Register with platform** for execution

The configuration serves as the contract between agent creator and platform. By submitting a configuration, the creator commits to specific preferences and goals. The platform stores the configuration and uses it throughout the agent's lifetime, ensuring consistency (agents can't change preferences midstream to game the mechanism).

AgentState:

```
public final class ImmutableAgentState {
    private final AgentId agentId;
    private final ImmutableResourceBundle currentResources;
    private final ImmutableList<Goal> goals;
    private final BigDecimal currencyBalance;

    // Runtime tracking
    private final ResourceRequest activeRequest;  // null if none
    private final ExecutionPlan activePlan;       // null if idle
    private final Instant lastAllocationTime;
    private final long totalAllocationsReceived;
    private final long totalJointAllocationsReceived;  // Track joint vs.
single

    // Private constructor
    private ImmutableAgentState(...) { ... }

    // State transition methods (return new state)
    public ImmutableAgentState withResources(ResourceBundle resources);
    public ImmutableAgentState withCurrencyBalance(BigDecimal balance);
    public ImmutableAgentState withActiveRequest(ResourceRequest request);

    // Query methods
    public boolean isGoalSatisfied(Goal goal);
    public double computeUtility(PreferenceFunction prefs);
}
```

AgentState is mutable in the sense that it evolves over time (resources acquired/released across multiple types, currency earned/spent, goals achieved), but each state version is immutable. When an agent receives a joint allocation (multiple resource types), we create a new AgentState with updated resources across all types rather than modifying the existing state piecemeal.

The runtime tracking fields (activeRequest, activePlan, timestamps, counts including joint allocation count) enable monitoring and debugging. We can query "how many allocations has agent X received?" or "when was agent Y last allocated resources?" or "what fraction of agent Z's allocations were joint vs. single-resource?" without maintaining separate logs. All information is embedded in the state.

Goal Representation:

Goals are threshold-based rather than maximization-based:

```
public interface Goal {
    boolean isSatisfied(ImmutableAgentState state);
    String getDescription();
}

public class MaintainThresholdGoal implements Goal {
    private final ResourceType resourceType;
    private final long threshold;

    @Override
    public boolean isSatisfied(ImmutableAgentState state) {
        return state.getCurrentResources().getQuantity(resourceType)
               >= threshold;
    }
}

public class AchieveThresholdGoal implements Goal {
    private final ResourceType resourceType;
    private final long threshold;
    private final long deadlineTimestep;

    @Override
    public boolean isSatisfied(ImmutableAgentState state) {
        long currentTimestep = state.getTimestep();
        if (currentTimestep > deadlineTimestep) {
            return false;  // Deadline passed without achievement
        }
        return state.getCurrentResources().getQuantity(resourceType)
               >= threshold;
    }
```

```
}
```

Goals provide compositional reasoning: we can check "are all agents' goals simultaneously satisfiable?" by solving a feasibility problem (find joint allocation where all goals satisfied across all resource types). For Milestone 1 with linear preferences and aggressive grouping, this is tractable. Goals also provide semantic meaning beyond numeric utilities—"agent needs compute ≥ 20" is more interpretable than "agent's utility function weights compute at 0.7."

4.4 Resource Type System

Resource types define what kinds of resources exist and their properties. For Milestone 1, resource types are enumerated (fixed set), but the architecture supports extension through plugin-style registration in Phase 2.

ResourceType Enum:

```
public enum ResourceType {
    // Computational resources
    COMPUTE_UNITS("compute_units", "Abstract CPU capacity", true, 1000),
    MEMORY_UNITS("memory_units", "Abstract RAM capacity", true, 500),
    STORAGE_BLOCKS("storage_blocks", "Persistent storage", true, 2000),
    GPU_HOURS("gpu_hours", "Accelerator time", true, 100),

    // Informational resources
    DATASET_ACCESS("dataset_access", "Dataset access quota", false, 10),
    MODEL_INFERENCE_CREDITS("model_inference_credits", "ML API quota",
true, 5000),
    API_CALL_QUOTA("api_call_quota", "External service quota", true, 1000),
    KNOWLEDGE_BASE_TOKENS("knowledge_base_tokens", "LLM context", true,
100000),

    // Physical resources (mocked for Milestone 1)
    LAB_TIME_SLOTS("lab_time_slots", "Equipment time", false, 50),
    SENSOR_ACCESS_WINDOWS("sensor_access_windows", "IoT device time",
false, 60),
    PHYSICAL_ROBOT_HOURS("physical_robot_hours", "Robot usage time", false,
30),
    PRINTER_3D_RESERVATIONS("3d_printer_reservations", "3D printer time",
false, 20),

    // Collaborative resources
```

```
    COLLABORATION_CREDITS("collaboration_credits", "Multi-agent coord",
true, 80),
    SHARED_WORKSPACE_HOURS("shared_workspace_hours", "Joint workspace",
false, 50),
    COMMUNICATION_BANDWIDTH("communication_bandwidth", "Inter-agent msg",
true, 10000);

    private final String identifier;
    private final String description;
    private final boolean divisible;  // Can be split into sub-units?
    private final long defaultPoolQuantity;

    ResourceType(String identifier, String description,
                 boolean divisible, long defaultPoolQuantity) {
        this.identifier = identifier;
        this.description = description;
        this.divisible = divisible;
        this.defaultPoolQuantity = defaultPoolQuantity;
    }

    // Natural ordering for lock acquisition (alphabetical by identifier)
    public int compareTo(ResourceType other) {
        return this.identifier.compareTo(other.identifier);
    }

    // Getters
}
```

Resource Categories:

The four categories (computational, informational, physical, collaborative) are conceptual groupings for human understanding, not architectural distinctions. The mechanism treats all resource types identically—the same Proportional Fairness optimization works for compute units, dataset access, lab time, and collaboration credits. When joint contentions span multiple categories (common with aggressive grouping), the mechanism optimizes across categories uniformly. This generality is a key strength: we don't need resource-type-specific allocation logic.

Natural Ordering for Lock Acquisition:

The `compareTo()` method provides a total ordering on resource types (alphabetical by identifier). This ordering is used by the locking hierarchy to prevent deadlock when acquiring

multiple resource locks. For joint allocations touching multiple resource types, locks are acquired in sorted order: if allocating {compute, storage, dataset}, acquire locks in that alphabetical order regardless of the sequence they appear in the allocation matrix.

Divisibility Property:

Some resources are naturally divisible (compute units can be split into fractional amounts), others are atomic (dataset access is binary—you have it or you don't). For Milestone 1, we discretize everything (quantities are integers), making divisibility irrelevant. Phase 2 with continuous resources will distinguish: divisible resources can have fractional allocations, atomic resources remain integer-only.

Default Pool Quantities:

Each resource type has a default initial quantity for the resource pool. These defaults enable quick platform setup without requiring detailed configuration. For production deployments, administrators override defaults based on actual hardware capacity (how many compute units does the cluster provide?) and policy (how much access should be shared?).

Extension Mechanism:

Phase 2 will replace the enum with a plugin-style registry:

```
public interface ResourceTypeRegistry {
    void registerType(ResourceTypeDescriptor descriptor);
    ResourceTypeDescriptor lookup(String identifier);
    Collection<ResourceTypeDescriptor> allTypes();
}
```

This allows adding new resource types without recompiling the platform. An administrator could define "quantum_compute_hours" with specific properties, register it, and agents can immediately request it. The mechanism code doesn't change—it works generically over any registered resource types.

4.5 Event Types and Their Payloads

Events represent state transitions or notifications. Each event type has a specific payload structure encoding the information needed to understand and process the event. Event payloads are enhanced to support joint contentions and allocations.

Event Hierarchy:

```
public sealed interface Event permits
    ResourceRequestEvent,
    ResourceRequestBatchEvent,
```

```
    ContentionDetectedEvent,
    ArbitrationCompleteEvent,
    AllocationEnforcedEvent,
    ResourceReleaseEvent,
    StateTransitionEvent {

    UUID getEventId();
    Instant getTimestamp();
    EventType getType();
}
```

Using sealed interfaces (Java 15+) makes the event taxonomy closed—only the listed event types exist. This prevents subsystems from creating ad-hoc event types without careful design.

ResourceRequestEvent:

```
public record ResourceRequestEvent(
    UUID eventId,
    Instant timestamp,
    AgentId agentId,
    Map<ResourceType, QuantityRange> requests
) implements Event {

    public record QuantityRange(long ideal, long minimum) {
        public QuantityRange {
            if (minimum > ideal) {
                throw new IllegalArgumentException(
                    "Minimum cannot exceed ideal"
                );
            }
        }
    }

    @Override
    public EventType getType() { return EventType.RESOURCE_REQUEST; }
}
```

This event represents an agent making a resource request, potentially for multiple resource types simultaneously. The QuantityRange nested record encodes both ideal (what agent wants) and minimum (what agent needs to function) for each type. The separation enables partial allocation across multiple types: platform can allocate between minimum and ideal per type, requiring agent to adapt but allowing progress.

ContentionDetectedEvent:

```
public record ContentionDetectedEvent(
    UUID eventId,
    Instant timestamp,
    Set<ResourceType> contestedResources,  // Multiple types for joint
    Set<AgentId> involvedAgents,           // Union across all types
    Map<ResourceType, Long> totalDemands,  // Per-type demand
    Map<ResourceType, Long> availableSupplies,  // Per-type supply
    Map<ResourceType, Double> contentionRatios,  // Per-type severity
    boolean isJoint                        // True if multiple resource
types
) implements Event {
    @Override
    public EventType getType() { return EventType.CONTENTION_DETECTED; }

    public int getProblemSize() {
        return involvedAgents.size() * contestedResources.size();
    }
}
```

This event signals that ContentionDetector found over-demand for one or more resource types. The `isJoint` flag and size of `contestedResources` indicate whether this is a joint contention (multiple types) or single-resource contention. The per-type demand and supply information enables analysis of which resources are most severely contested and might benefit from targeted interventions (increasing pool quantities, encouraging releases).

ArbitrationCompleteEvent:

```
public record ArbitrationCompleteEvent(
    UUID eventId,
    Instant timestamp,
    Set<ResourceType> allocatedResourceTypes,  // Single or multiple
    Map<AgentId, Map<ResourceType, Long>> allocationMatrix,  // Full matrix
    Map<AgentId, BigDecimal> currencyBurned,
```

```
    Map<AgentId, Double> priorityWeights,
    double objectiveValue,
    long computationTimeMs,
    boolean isJoint,
    SolverMetadata solverDetails
) implements Event {
    @Override
    public EventType getType() { return EventType.ARBITRATION_COMPLETE; }
}
```

This event captures joint arbitration results. The `allocationMatrix` now stores the full agent × resource type allocation, enabling allocations across multiple types to be represented. The event includes **currencyBurned** (not payments), reflecting the burn-not-pay Priority Economy model. The priorityWeights field enables verification: did the arbitrator use the weights we expect (BaseWeight + burned)? The solverDetails provide debugging information (iterations, convergence status, numerical residuals) particularly valuable for large joint contentions.

AllocationEnforcedEvent:

```
public record AllocationEnforcedEvent(
    UUID eventId,
    Instant timestamp,
    Map<AgentId, ResourceBundle> allocations,  // Full bundles per agent
    ImmutableResourceLedger newLedgerState,
    boolean isJoint,
    Set<ResourceType> resourceTypesAllocated
) implements Event {
    @Override
    public EventType getType() { return EventType.ALLOCATION_ENFORCED; }
}
```

This event marks the commit point where allocations become actual ownership. For joint allocations, this represents the atomic commit of all resource types together (Property 6.3). The newLedgerState field provides the complete post-allocation ledger for verification and auditing.

ResourceReleaseEvent:

```
public record ResourceReleaseEvent(
    UUID eventId,
    Instant timestamp,
```

```
    AgentId agentId,
    ResourceBundle releasedResources,  // May contain multiple types
    Map<ResourceType, Instant> originalLeaseExpiries,  // Per type
    Map<ResourceType, BigDecimal> earningsPerType,  // Breakdown
    BigDecimal totalCurrencyEarned,  // Sum across types
    String releaseReason
) implements Event {
    @Override
    public EventType getType() { return EventType.RESOURCE_RELEASE; }
}
```

This event captures resource releases with earning information broken down by resource type. The `earningsPerType` field shows how much currency was earned from releasing each type (useful for analysis of which resource releases are most lucrative). The `originalLeaseExpiries` field enables TimeRemaining calculation for verification. The releaseReason ("task_complete", "lease_expired", "voluntary_early_release") helps diagnose agent behavior—are agents releasing promptly or waiting until forced?

---

5. Execution Model and Agent Lifecycle

Agents in our system are not independent processes in the operating system sense—they're managed entities that execute within platform-controlled contexts. This section explains the agent lifecycle (registration, activation, execution, termination), the AgentBehavior interface contract, and how the platform manages agent execution with sandboxing and timeouts.

5.1 Agent Lifecycle States

An agent progresses through several states during its lifetime on the platform. The state machine reflects an understanding of how joint allocations affect the parts of the lifecycle relevant to the arbitrator:

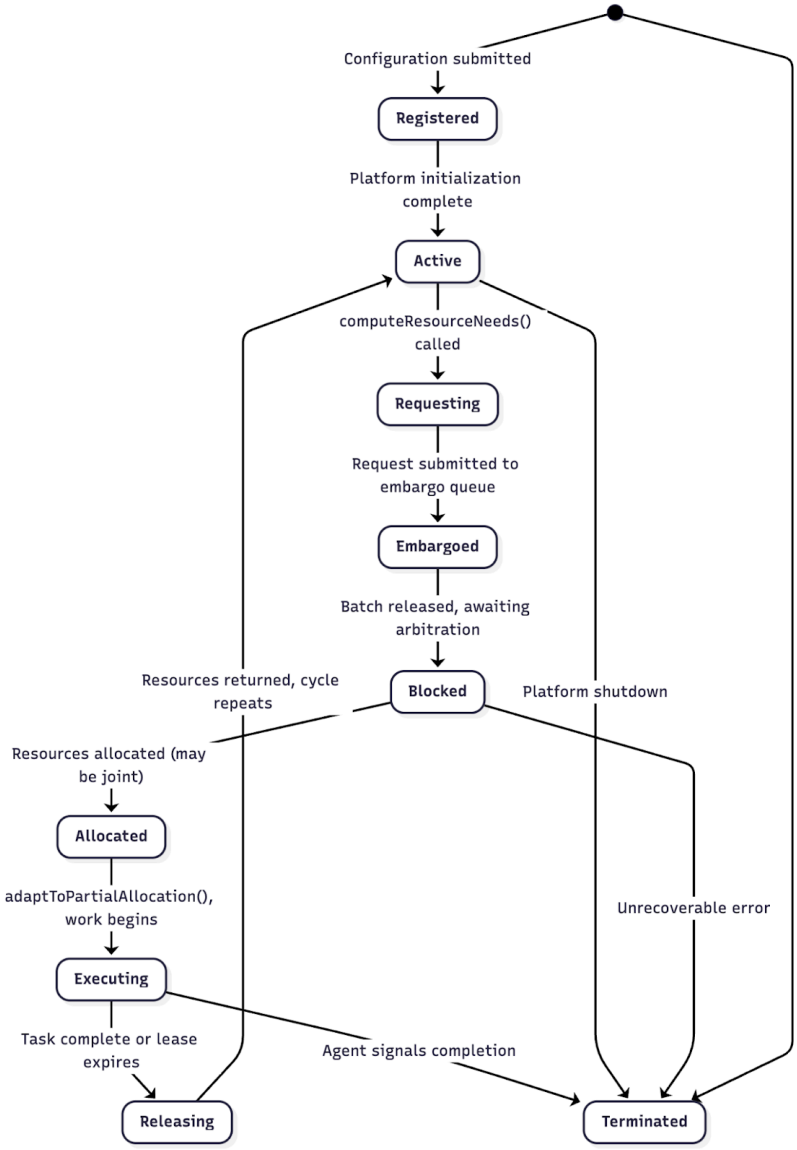

Figure 3. Agent Lifecycle State Machine

State Descriptions:

Registered: Agent configuration has been submitted and validated, but execution hasn't started. The platform has created an AgentExecutionContext for this agent but hasn't yet called any behavioral methods. This state exists during platform initialization before the main event loop begins.

Active: Agent is running and can make resource requests. The platform may periodically call `computeResourceNeeds()` to understand current desires. In this state, the agent has no pending requests and is idle or doing work with currently-held resources.

Requesting: Agent has computed resource needs and submitted ResourceRequestEvent (potentially requesting multiple resource types). The request is in the embargo queue waiting for batch processing. The agent cannot make new requests while one is pending (prevents spam).

Embargoed: Request is in the embargo queue, waiting for the 100ms window to expire. Other agents' requests may arrive during this window and be batched together.

Blocked: Request has been released from embargo queue and processed. Contention detected (possibly joint contention involving multiple resource types). Agent is blocked waiting for arbitration to complete. The agent cannot proceed with work until resources are allocated.

Allocated: Arbitration complete, resources allocated (possibly across multiple types from joint optimization), agent has been notified. The platform calls `adaptToPartialAllocation()` to give agent opportunity to adjust execution plan based on received resources (which may be less than ideal, across one or more resource types).

Executing: Agent is performing work with allocated resources. It may run for extended period (minutes) in this state. The platform doesn't interrupt execution unless leases expire or agent voluntarily signals completion.

Releasing: Agent has finished work or leases expired. Resources are being transferred back to pool (across all types in the original allocation bundle), currency earnings are being computed and minted. This state is brief (milliseconds) but logically distinct.

Terminated: Agent has completed its goals and exited, or platform has shut down, or agent encountered unrecoverable error. No further behavioral methods will be called. The agent's final state is preserved in event history for auditing.

State Transitions:

Most transitions are initiated by the agent (requesting resources, completing work) or by the platform (allocating resources, expiring leases). The agent has bounded autonomy: it controls when to request and when to release, but not when allocations occur (that's platform-mediated arbitration) or when leases expire (that's platform-configured policy).

The cycle from Active → Requesting → Embargoed → Blocked → Allocated → Executing → Releasing → Active can repeat many times as an agent performs multiple tasks. Each cycle represents one request-allocate-execute-release sequence. Long-running agents might go through hundreds of cycles during their lifetime.

5.2 AgentBehavior Interface Contract

The AgentBehavior interface defines the methods that agent implementations must provide. The platform calls these methods at specific lifecycle points, always passing immutable state snapshots to prevent information leakage or state corruption.

```
public interface AgentBehavior {
    /**
     * Compute current resource needs based on state.
     * May request multiple resource types.
     * Called periodically by platform (every ~5-10 seconds) or when
```

```
     * agent transitions to Active state.
     *
     * @param myState Immutable snapshot of this agent's current state
     * @param worldState Immutable snapshot of observable global state
     * @return Resource request with ideal and minimum quantities per type
     * @throws AgentException if computation fails
     */
    ResourceRequest computeResourceNeeds(
        ImmutableAgentState myState,
        ImmutableGlobalState worldState
    ) throws AgentException;

    /**
     * Adapt execution plan to partial allocation.
     * May receive partial allocations across multiple resource types.
     * Called immediately after allocation when agent receives
     * less than ideal quantity for any resource type.
     *
     * @param requested What agent originally requested (may be
multi-resource)
     * @param received What agent actually got (min ≤ received ≤ ideal per
type)
     * @return Execution plan for partial resources, or CANNOT_PROCEED
     */
    ExecutionPlan adaptToPartialAllocation(
        ResourceRequest requested,
        ResourceAllocation received
    );

    /**
     * Provide predictive signal for future needs (optional).
     * Helps platform with proactive arbitration by anticipating
     * future contentions. Can return null if no prediction available.
     *
     * @param myState Current agent state
     * @param horizonTimesteps How many timesteps ahead to predict
     * @return Predicted needs across resource types, or null
     */
```

```
    @Nullable
    PredictiveResourceSignal predictFutureNeeds(
        ImmutableAgentState myState,
        int horizonTimesteps
    );

    /**
     * Evaluate utility of observable state.
     * Called by platform during arbitration to compute utilities
     * for Proportional Fairness objective. Must match declared
     * preference function from configuration.
     *
     * @param state Observable state to evaluate
     * @return Utility value (must equal Σ w_i · r_i for linear prefs)
     */
    double evaluateUtility(ImmutableObservableState state);

    /**
     * Decide how much currency to burn for priority.
     * Called during contention detection, before arbitration.
     * Agent has opportunity to signal urgency by committing currency.
     *
     * @param myState Current state including currency balance
     * @param worldState Global state including current contention
     * @param currentNeed The resource request being prioritized
     * @return Amount of currency to burn (0 to current balance)
     */
    BigDecimal decidePriorityBudget(
        ImmutableAgentState myState,
        ImmutableGlobalState worldState,
        ResourceRequest currentNeed
    );
}
```

Method Responsibilities and Semantics:

computeResourceNeeds() is the primary decision method where agents express what they want across multiple resource types. The agent examines its current state (what resources does

it have across all types? what goals remain? what is its progress?) and global state (what's the current contention like? what are other agents requesting?) and returns a ResourceRequest specifying ideal and minimum quantities for each resource type it's interested in.

This method is called periodically (every 5-10 seconds) even when the agent has resources, allowing it to adjust requests dynamically. If circumstances change (a goal is achieved, new tasks arrive, resource needs evolve), the agent can request different resources in the next call. This periodic polling is simple but sufficient for Milestone 1's scale. Phase 2 might add event-driven requesting where agents can trigger requests immediately when needs change.

The method must complete within timeout (200ms default). If it exceeds timeout, the platform interrupts it and uses the agent's previous request (if any) or a default empty request. Agents should compute needs quickly, avoiding complex optimization or extensive simulation within this method.

adaptToPartialAllocation() handles the case where the agent receives less than ideal allocation across one or more resource types. For example, agent requests {compute:100, storage:50} with minimums {compute:50, storage:20}, receives {compute:70, storage:35}. The agent received partial allocations for both types and must decide: can I proceed with 70 compute and 35 storage? If yes, how do I adapt my execution plan (scale down workload, reduce quality, take longer)? If no, return CANNOT_PROCEED.

This method is critical for graceful degradation under resource scarcity. Without partial allocation, the platform would face all-or-nothing decisions (either give agent full request across all types or give nothing). With partial allocation and adaptation, the platform can allocate partial amounts across multiple types (leaving resources for other agents), and agents proceed with reduced workloads. Total system throughput is higher.

The adaptation logic is agent-specific. A machine learning training agent might scale batch size proportionally to compute allocated while using storage for checkpointing. A data processing agent might reduce quality (lossy compression instead of lossless) proportionally across both compute and storage constraints. A collaborative agent might reduce collaboration session size based on available collaboration_credits and workspace_hours. The platform doesn't dictate how to adapt; it just requires that agents implement adaptation logic that can handle multi-resource partial allocations.

decidePriorityBudget() is where agents engage in intertemporal strategy. Given current currency balance, current need urgency across potentially multiple resource types, and expected future opportunities, how much currency should the agent burn now? The decision considers that burning currency provides priority across ALL contested resource types in a joint contention—one currency expenditure affects multiple resource allocations simultaneously when aggressive grouping merges contentions.

This creates interesting strategic considerations: if an agent knows their request will likely be grouped with others into a joint contention (because they're requesting compute, which many agents want), burning currency provides leverage across all resource types in that joint contention, not just compute. An agent might burn 50 currency to get priority for compute, and as a side effect, also get priority for storage and dataset in the joint optimization.

The method is called just before arbitration when contention is detected. The agent sees the ContentionDetectedEvent (how many agents competing? how many resource types involved? is

this joint?) and can adjust strategy accordingly. If contention is joint and severe (many agents, many resources, high ratios), burning more currency is more impactful (competition is fierce across multiple dimensions). If contention is single-resource and mild (few agents, one resource, low ratio), burning even small amounts provides good advantage.

evaluateUtility() must match the declared preference function across all resource types. If agent's configuration specifies `w_compute=0.5, w_storage=0.3, w_dataset=0.2`, then `evaluateUtility(state)` must return `0.5 * state.compute + 0.3 * state.storage + 0.2 * state.dataset`. The platform uses this method during arbitration to compute $\Phi_i(A)$ values for the Proportional Fairness objective across all resource types. If the agent returns values inconsistent with declared preferences, mechanism properties are not guaranteed.

Method Call Timing and Threading:

The platform calls AgentBehavior methods from its executor thread pool, not from the agent's "own" thread (agents don't have dedicated threads in Milestone 1). When the platform needs to call a method:

1. **Submit to executor:** Create callable that invokes the method with immutable state snapshots
2. **Set timeout:** Use `Future.get(timeout)` to enforce time limit (200ms)
3. **Execute:** Thread from pool executes the callable
4. **Collect result:** If completes within timeout, return result; if exceeds timeout, cancel and throw AgentTimeoutException
5. **Release thread:** Thread returns to pool for reuse

This model means multiple agents might have methods executing concurrently (each on different pool thread), but no single agent has multiple methods executing simultaneously (one call per agent at a time, enforced by platform synchronization).

5.3 AgentExecutionManager: Lifecycle Management

The AgentExecutionManager handles agent registration, behavioral code execution, timeout enforcement, and lifecycle transitions. It maintains AgentExecutionContext objects that wrap each agent's configuration and runtime state.

Responsibilities:

Registration: When agent configurations are submitted, the manager validates them (preferences sum to 1.0, goals are coherent, behavior class can be loaded), instantiates the behavior class, creates AgentExecutionContext, and adds to its registry.

Execution: When the platform needs to call an agent's behavioral method (computeResourceNeeds, adaptToPartialAllocation, decidePriorityBudget), the manager provides thread pool execution with timeout enforcement, defensive state copying, exception handling, and result collection.

Sandboxing: The manager enforces execution constraints: timeouts (200ms default), memory limits (via monitoring, not hard enforcement in Milestone 1), no inter-agent communication

(enforced architecturally by not providing channels), no external I/O (Phase 2 adds classloader isolation; Milestone 1 relies on agent behavior not attempting I/O).

Monitoring: The manager tracks agent execution metrics: how many timeouts has agent experienced? how long do methods typically take? what fraction of allocations were joint vs. single-resource? are there patterns indicating misbehavior?

Interface:

```
public interface AgentExecutionManager {
    /**
     * Register agent with platform.
     * Validates configuration, instantiates behavior, creates context.
     *
     * @param config Agent configuration to register
     * @throws ConfigurationException if config invalid or behavior class
unloadable
     */
    void registerAgent(AgentConfiguration config)
        throws ConfigurationException;

    /**
     * Execute agent's computeResourceNeeds with timeout and sandboxing.
     *
     * @param agentId Agent to query
     * @param state Current global state (immutable snapshot)
     * @return Resource request (may span multiple types), or default if
timeout
     * @throws AgentExecutionException if execution fails
     */
    ResourceRequest executeComputeNeeds(
        AgentId agentId,
        ImmutableGlobalState state
    ) throws AgentExecutionException;

    /**
     * Execute agent's decidePriorityBudget with timeout.
     *
     * @param agentId Agent to query
     * @param state Current state
```

```
     * @param need Current resource need (may be multi-resource)
     * @return Currency amount to burn (validated against balance)
     */
    BigDecimal executePriorityBudget(
        AgentId agentId,
        ImmutableGlobalState state,
        ResourceRequest need
    ) throws AgentExecutionException;

    /**
     * Get all active agent contexts (for contention detection).
     */
    Collection<AgentExecutionContext> getActiveAgents();

    /**
     * Notify agent of allocation (async).
     * For joint allocations, notification includes full resource bundle.
     * Agent processes notification in background, prepares execution plan.
     */
    void notifyAllocation(AgentId agentId, ResourceAllocation allocation);
}
```

AgentExecutionContext:

Each registered agent has an AgentExecutionContext that serves as a container for agent-specific state and the behavior instance:

```
public class AgentExecutionContext {
    private final AgentId agentId;
    private final AgentConfiguration configuration;
    private final AgentBehavior behavior;  // instantiated behavior object
    private final long timeoutMs;

    // Runtime tracking
    private volatile ResourceRequest currentRequest;  // null if none
    private volatile ExecutionPlan currentPlan;      // null if idle
    private volatile Instant lastAllocationTime;
    private final AtomicLong totalMethodCalls;
```

```
    private final AtomicLong totalTimeouts;
    private final AtomicLong totalAllocationsReceived;
    private final AtomicLong totalJointAllocationsReceived;

    // Query methods
    public boolean hasActiveRequestFor(ResourceType type);
    public ResourceRequest getActiveRequest();
    public double getJointAllocationRate() {
        long total = totalAllocationsReceived.get();
        long joint = totalJointAllocationsReceived.get();
        return total > 0 ? (double) joint / total : 0.0;
    }

    // Getters and state management methods
}
```

The context provides thread-safe access to agent state (using volatile for single-field updates, atomic operations for counters) without requiring complex locking. The behavior instance is created once at registration time and reused for all method calls—we don't create a new instance per call because behavior objects may have internal state (learned patterns, cached computations) that persists across calls.

The new `totalJointAllocationsReceived` counter tracks how many of the agent's allocations came from joint arbitrations versus single-resource arbitrations. This metric helps understand how aggressive grouping affects individual agents and whether certain agents benefit more from cross-resource trades.

Timeout Enforcement Pattern:

The manager uses Java's `Future` mechanism for timeout enforcement:

```
public ResourceRequest executeComputeNeeds(
        AgentId agentId,
        ImmutableGlobalState state) {

    AgentExecutionContext ctx = agents.get(agentId);

    // Create defensive state copies
    ImmutableAgentState agentState = state.getAgentState(agentId);
    ImmutableGlobalState worldState = state; // already immutable
```

```
    // Submit to executor with timeout
    Future<ResourceRequest> future = executor.submit(() -> {
        return ctx.getBehavior().computeResourceNeeds(
            agentState,
            worldState
        );
    });

    try {
        // Block with timeout
        ResourceRequest result = future.get(timeoutMs,
TimeUnit.MILLISECONDS);
        ctx.incrementMethodCalls();
        return result;

    } catch (TimeoutException e) {
        // Cancel execution, record timeout, return fallback
        future.cancel(true);  // interrupt if possible
        ctx.incrementTimeouts();

        return ctx.getCurrentRequest() != null
            ? ctx.getCurrentRequest()  // reuse previous request
            : ResourceRequest.empty(); // or empty request

    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new AgentExecutionException("Interrupted", e);

    } catch (ExecutionException e) {
        // Agent code threw exception
        throw new AgentExecutionException(
            "Agent behavior threw exception",
            e.getCause()
        );
    }
}
```

This pattern provides several robustness properties that apply uniformly whether agents are requesting single resources or multiple resources in anticipation of joint allocation.

---

6. Event System and Communication

The event system provides the primary communication mechanism between platform components. Rather than components calling each other directly (tight coupling that creates brittle dependencies), they communicate through events (loose coupling that enables independent evolution). This section details the event bus architecture, embargo queue mechanics, and event handler patterns. ContentionDetectedEvent and ArbitrationCompleteEvent support joint contentions.

6.1 EventBus Architecture

The EventBus implements a publish-subscribe pattern where producers publish events without knowing who will consume them, and consumers subscribe to event types they're interested in without knowing who produces them. This inversion of control creates a plugin-like architecture where components can be added, removed, or replaced without affecting others.

Design Rationale:

Direct method calls create compile-time dependencies: if ComponentA calls ComponentB.method(), then A depends on B. Adding a new component C that should also react to what A does requires modifying A to call C.method() too. As the system grows, A accumulates dependencies on B, C, D, E, creating a coupling tangle that makes changes risky.

Event-based communication decouples producers from consumers: A publishes Event, and B, C, D subscribe to that event type. A doesn't know (or care) who's listening. Adding E just requires E to subscribe; no changes to A. Removing D just requires D to unsubscribe; no changes to anyone else. The EventBus serves as the mediator that routes events from publishers to subscribers.

Core Operations:

Publish: Post event to bus for distribution to subscribers. Most events are delivered synchronously (subscribers' handlers called before publish() returns), ensuring ordering guarantees. ResourceRequestEvent is special-cased through embargo queue for fairness.

Subscribe: Register handler for specific event type. Handler is a callback function invoked when events of that type are published. Multiple handlers can subscribe to the same event type (multicast).

Retrieve: Main platform event loop calls getNextEvent() to retrieve events from queue for processing. This supports the pattern where some events trigger immediate subscriber

reactions (synchronous publication) while others are queued for batch or deferred processing (asynchronous via queue).

Flow Diagram for Joint Contention:

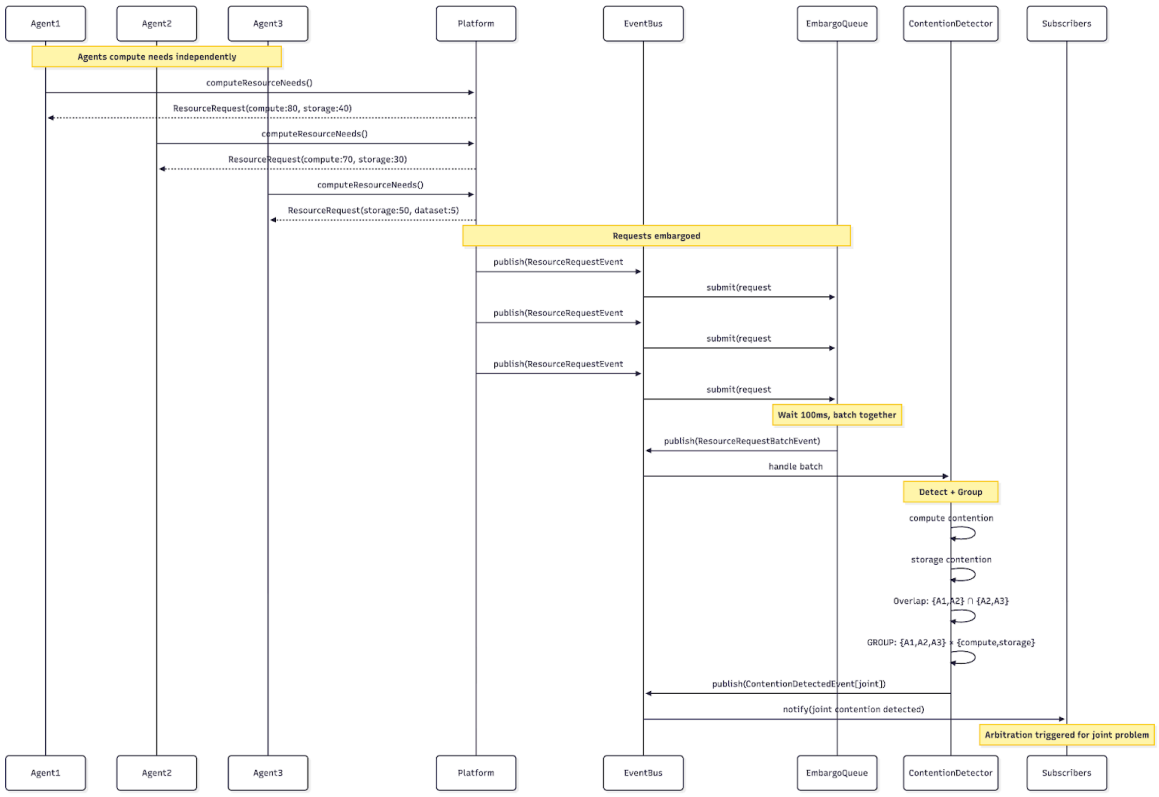

Figure 4. Sequence Diagram: Joint Contention Event Flow

EventBus Interface:

```
public interface EventBus {
    /**
     * Publish event to subscribers.
     * ResourceRequestEvent goes through embargo queue (async).
     * Other events delivered synchronously to subscribers.
     */
    void publish(Event event);

    /**
     * Subscribe to events of specific type.
     * Handler invoked synchronously when events published.
     */
    <T extends Event> void subscribe(
```

```
        Class<T> eventType,
        EventHandler<T> handler
    );

    /**
     * Unsubscribe handler from event type.
     */
    <T extends Event> void unsubscribe(
        Class<T> eventType,
        EventHandler<T> handler
    );

    /**
     * Retrieve next event from queue (blocking).
     * Used by platform main loop for deferred processing.
     */
    Event getNextEvent() throws InterruptedException;

    /**
     * Retrieve next event with timeout.
     * Returns null if timeout expires without event.
     */
    Event getNextEvent(long timeout, TimeUnit unit)
        throws InterruptedException;
}
```

Subscription Registry:

Internally, the EventBus maintains a mapping from event types to lists of handlers:

```
private final Map<Class<? extends Event>, List<EventHandler<?>>>
    subscriptions = new ConcurrentHashMap<>();
```

When `subscribe(EventType.class, handler)` is called, the handler is added to the list for that event type. When `publish(event)` is called, the bus looks up handlers for event.getClass() and invokes each handler's handle() method.

The ConcurrentHashMap provides thread-safe concurrent access—multiple threads can publish events or subscribe handlers simultaneously without external synchronization. The list of handlers for each type is copied-on-write (create new list when adding/removing handlers) to enable safe concurrent iteration during event publication.

Synchronous vs Asynchronous Publication:

Most events use **synchronous publication**: when publish(event) is called, all registered handlers are invoked immediately on the calling thread, and publish() doesn't return until all handlers complete. This provides strong ordering guarantees: if thread publishes Event A then Event B, and component C subscribes to both, C sees A before B (causality preserved).

The synchronous model also simplifies error handling: if a handler throws exception, it propagates to the publisher, who can decide how to handle it (log, retry, fail fast). This is preferable to asynchronous models where exceptions might be lost or delayed.

ResourceRequestEvent uses **asynchronous publication** via embargo queue: publish() submits to queue and returns immediately. The event is held for embargo window (100ms), then batch-released with other requests received during that window. This asynchrony is necessary for fairness (eliminate timing advantages) but is the exception, not the rule.

6.2 EmbargoQueue: Fair Request Batching

The EmbargoQueue implements the embargo mechanism that ensures fair request processing regardless of network latency or agent scheduling timing. Without embargo batching, agents with lower latency (faster network, closer to platform, luckier scheduling) would systematically receive priority through timing advantages. The embargo queue eliminates these advantages by batching near-simultaneous requests. The queue operates identically whether requests are for single resources or multiple resources—it batches by time window, not by resource type.

Mechanism:

When ResourceRequestEvent is published:
1. **Timestamp event:** Record exact time of publication (System.currentTimeMillis())
2. **Add to pending queue:** Place in concurrent queue of timestamped requests
3. **Return immediately:** Don't block publisher waiting for processing

Periodically (every 100ms by default):
1. **Collect expired requests:** Scan pending queue, collect all requests older than embargo window
2. **Create batch:** Package collected requests into ResourceRequestBatchEvent
3. **Publish batch:** Send batch to event bus for synchronous distribution to ContentionDetector
4. **Remove from queue:** Collected requests removed from pending

Why τ=100ms:

The embargo window duration is a tradeoff between fairness and latency:

Shorter windows (τ=50ms) reduce agent wait time but provide less batching (fewer requests collected together, less fairness protection against timing variations).

Longer windows (τ=200ms) improve batching (more requests collected, better fairness) but increase latency (agents wait longer for responses).

For Milestone 1, τ=100ms balances these concerns. Typical network jitter is 10-50ms, so 100ms batches most near-simultaneous requests. Response latency is 100ms (embargo) + 50-500ms (arbitration, depending on joint contention size) = 150-600ms total, acceptable for interactive coordination.

Phase 2 might add **adaptive embargo windows:** expand during high contention (when fairness matters most, many requests arriving, longer batching improves fairness) and contract during low contention (when latency matters more, few requests, batching less critical).

Implementation Pattern:

```
public class EmbargoQueue {
    private final long embargoWindowMs;
    private final ConcurrentLinkedQueue<TimestampedRequest> pending;
    private final ScheduledExecutorService batchProcessor;
    private final EventBus eventBus;

    public EmbargoQueue(long embargoWindowMs, EventBus eventBus) {
        this.embargoWindowMs = embargoWindowMs;
        this.pending = new ConcurrentLinkedQueue<>();
        this.eventBus = eventBus;
        this.batchProcessor = Executors.newSingleThreadScheduledExecutor();

        // Start periodic batch processing
        batchProcessor.scheduleAtFixedRate(
            this::processBatch,
            embargoWindowMs,  // initial delay
            embargoWindowMs,  // period
            TimeUnit.MILLISECONDS
        );
    }

    public void submit(ResourceRequestEvent request) {
        pending.offer(new TimestampedRequest(
            request,
```

```
            System.currentTimeMillis()
        ));
    }

    private void processBatch() {
        long now = System.currentTimeMillis();
        long cutoff = now - embargoWindowMs;

        List<ResourceRequestEvent> batch = new ArrayList<>();

        // Collect expired requests
        Iterator<TimestampedRequest> iter = pending.iterator();
        while (iter.hasNext()) {
            TimestampedRequest tr = iter.next();
            if (tr.timestamp <= cutoff) {
                batch.add(tr.request);
                iter.remove();
            }
        }

        // Publish batch if non-empty
        if (!batch.isEmpty()) {
            eventBus.publish(new ResourceRequestBatchEvent(
                batch,
                cutoff,  // embargo window start
                now      // embargo window end
            ));
        }
    }
}
```

The embargo queue handles multi-resource requests identically to single-resource requests. A request for {compute:80, storage:40} is timestamped and batched just like a request for {compute:80}. The ContentionDetector receives the batch and performs grouping analysis based on which resource types are contested and which agents overlap, regardless of how many resource types each agent requested.

6.3 Event Handler Patterns

Event handlers are the callbacks that components implement to react to events. Handlers must follow specific patterns to ensure correctness and avoid subtle concurrency bugs.

Handler Interface:

```
@FunctionalInterface
public interface EventHandler<T extends Event> {
    /**
     * Handle event. Called synchronously by EventBus on publishing thread.
     * Should complete quickly (< 10ms for most events, < 200ms for
complex).
     * Must not throw unchecked exceptions (catch and log instead).
     * For joint contentions, may take longer (200-500ms for large
problems).
     */
    void handle(T event);
}
```

Handler Patterns and Anti-Patterns:

Pattern: Quick Processing

Handlers should complete quickly because they're invoked synchronously on publishing thread. If handler blocks for long time (waiting on I/O, doing expensive computation), it blocks the publisher. For expensive operations like joint arbitration (which may take 200-500ms for large problems), handler should submit work to executor and return immediately:

```
public class ArbitrationHandler implements
EventHandler<ContentionDetectedEvent> {
    private final ExecutorService executor;
    private final ProportionalFairnessArbitrator arbitrator;

    @Override
    public void handle(ContentionDetectedEvent event) {
        // For joint contentions, arbitration may take 200-500ms
        // Submit to executor to avoid blocking publisher
        executor.submit(() -> {
            try {
                AllocationResult result = arbitrator.arbitrate(
```

```
                    event.getContention(),
                    getCurrentState()
                );
                eventBus.publish(new ArbitrationCompleteEvent(result));
            } catch (ArbitrationException e) {
                logger.error("Joint arbitration failed", e);
                eventBus.publish(new ArbitrationFailedEvent(
                    event.getContention(), e
                ));
            }
        });
    }
}
```

Pattern: Exception Safety

Handlers must not throw unchecked exceptions that propagate to event bus. If handler might fail, catch exceptions and log:

```
@Override
public void handle(ArbitrationCompleteEvent event) {
    try {
        // For joint allocations, enforcement involves multiple resource
types
        resourceManager.enforceAllocation(event.getAllocation());
    } catch (Exception e) {
        logger.error("Failed to enforce joint allocation", e);
        // Optionally publish error event for platform to handle
        eventBus.publish(new AllocationFailedEvent(event, e));
    }
}
```

Subscription Example for Joint Contention Flow:

Setting up event subscriptions during platform initialization:

```
// ContentionDetector subscribes to resource request batches
eventBus.subscribe(
```

```
    ResourceRequestBatchEvent.class,
    batch -> {
        // Detect single-resource contentions, then group
        Set<Contention> contentions = contentionDetector.detectAndGroup(
            batch,
            getCurrentState()
        );
        // Publish each contention (may be joint)
        contentions.forEach(c ->
            eventBus.publish(new ContentionDetectedEvent(c))
        );
    }
);

// Arbitrator subscribes to contention events (handles both single and
joint)
eventBus.subscribe(
    ContentionDetectedEvent.class,
    contentionEvent -> {
        Contention c = contentionEvent.getContention();
        // Arbitrator handles single-resource and joint uniformly
        AllocationResult result = arbitrator.arbitrate(c, currentState);
        eventBus.publish(new ArbitrationCompleteEvent(result));
    }
);

// ResourceManager subscribes to arbitration results
eventBus.subscribe(
    ArbitrationCompleteEvent.class,
    arbitrationEvent -> {
        // Enforcement uses TransactionManager for atomicity
        GlobalState newState = resourceManager.enforceAllocation(
            arbitrationEvent.getAllocation(),
            currentState
        );
        updateState(newState);
        eventBus.publish(new AllocationEnforcedEvent(newState));
    }
```

```
);
```

This subscription pattern creates the flow: ResourceRequest → Batch → DetectAndGroup → Contention(joint) → Arbitrate → Enforce(atomic), with each component subscribing to the event type it cares about and publishing events for next stage.

7. Resource Management and the Priority Economy

Resource management encompasses two interrelated concerns: tracking resource ownership (who has what resources, ensuring conservation across all types) and managing the Priority Economy (agents earning currency through releases, spending through burning, balances equilibrating over time). This section specifies the ResourceManager component and explains how it coordinates with PriorityWeightCalculator and TransactionManager to handle both single-resource and multi-resource operations atomically.

7.1 ResourceManager: Ownership and Transfers

The ResourceManager is responsible for enforcing allocations computed by the arbitrator, tracking resource leases across multiple resource types, handling releases, and maintaining resource conservation invariants. For joint allocations involving multiple resource types and agents, the ResourceManager ensures atomic enforcement—either all resources for all agents are allocated together, or none are (Property 6.3). This atomic enforcement is critical for maintaining system consistency when contentions are grouped aggressively.

Core Responsibilities:

Joint allocation enforcement: When arbitrator computes joint allocation (spanning multiple resource types and agents), ResourceManager coordinates with TransactionManager to apply all changes atomically: transfers resources across all types for all agents, burns currency from all agents, records lease expirations for all allocated resources, all in one atomic transaction. If any step fails (insufficient resources discovered after formulation, invariant violation, system error), all changes roll back and no agent receives partial allocations.

Release processing: When agents release resources (potentially multiple types from joint allocations), ResourceManager transfers resources from agent back to pool across all types, calculates earnings summed across all released types using the Priority Economy formula, mints currency into agent balance as a single atomic operation.

Lease tracking: Maintains map of when each agent's resource leases expire (per resource type), enables automatic reclamation when leases expire (resources released back to pool if agent hasn't done so voluntarily), provides data for TimeRemaining calculation in earnings formula.

Conservation enforcement: Before every state transition involving resources, validates that resource conservation holds for ALL resource types (total quantity unchanged per type), rejects

transitions that would violate conservation for any type. This validation is particularly important for joint allocations where multiple types are transferred simultaneously.

Integration extensibility: Phase 2 semantic service integration requires minimal ResourceManager changes. Service credits are tracked in the ResourceLedger alongside computational resources, with the same conservation invariants (`Σ_agents service_credits + pool_credits = initial_credits`). The manager doesn't distinguish service from computational resources in its enforcement logic; services are additional resource types in the ledger with identical accounting semantics.

Interface:

```
public interface ResourceManager {
    /**
     * Enforce allocation by updating ledger atomically.
     * For joint allocations, transfers ALL resource types for ALL agents atomically.
     * Transfers resources from pool to agents, burns currency, sets leases.
     * Uses TransactionManager to ensure atomicity (Property 6.3).
     *
     * @param result Allocation to enforce (may span multiple resource types)
     * @param currentState Current state to transition from
     * @return New state with all allocations applied atomically
     * @throws InsufficientResourcesException if allocation exceeds availability for any type
     * @throws TransactionException if atomic commit fails
     */
    GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState
    ) throws InsufficientResourcesException, TransactionException;

    /**
     * Process resource release (potentially multiple types).
     * Transfers resources back to pool, calculates and mints earnings.
     * Earnings computed per type using Priority Economy formula, then summed.
     *
     * @param agentId Agent releasing resources
     * @param resources Bundle being released (may contain multiple types)
```

```
     * @param currentState Current state
     * @return New state with releases applied and total earnings minted
     */
    GlobalState releaseResources(
        AgentId agentId,
        ResourceBundle resources,
        GlobalState currentState
    );

    /**
     * Check if allocation is feasible given current pool state.
     * Validates across ALL resource types in allocation.
     * Used for pre-validation before enforcement.
     *
     * @param result Allocation to check (may be joint)
     * @param state Current state
     * @return true if pool has sufficient quantities for all types
     */
    boolean isAllocationFeasible(
        AllocationResult result,
        GlobalState state
    );

    /**
     * Get lease expiration times for agent's resource holdings.
     * Returns per-resource-type expiration map.
     *
     * @param agentId Agent to query
     * @param state Current state
     * @return Map from resource types held to expiration times
     */
    Map<ResourceType, Instant> getLeaseExpirations(
        AgentId agentId,
        GlobalState state
    );

    /**
```

```
     * Check for expired leases and automatically reclaim resources.
     * Called periodically by platform monitoring thread.
     *
     * @param state Current state
     * @return New state with expired resources reclaimed
     */
    GlobalState reclaimExpiredLeases(GlobalState state);
}
```

Atomic Enforcement Process for Joint Allocations:

When enforceAllocation() is called with a joint AllocationResult (multiple resource types, multiple agents), the enforcement process must ensure atomicity as specified in Property 6.3:

```
public GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState)
        throws InsufficientResourcesException, TransactionException {

    // Pre-validation: Check feasibility across all resource types
    if (!isAllocationFeasible(result, currentState)) {
        throw new InsufficientResourcesException(
            "Insufficient resources for joint allocation across types: " +
            result.getAllocatedResourceTypes()
        );
    }

    // Pre-validation: Check all agents have sufficient currency
    for (Map.Entry<AgentId, BigDecimal> entry :
            result.getCurrencyBurned().entrySet()) {
        BigDecimal balance = currentState.getAgentState(entry.getKey())
            .getCurrencyBalance();
        if (balance.compareTo(entry.getValue()) < 0 &&
            balance.compareTo(BigDecimal.valueOf(-100)) < 0) {
            throw new InsufficientCurrencyException(
                "Agent " + entry.getKey() + " cannot afford to burn " +
                entry.getValue()
            );
```

```
        }
    }

    // Begin atomic transaction
    Transaction txn = transactionManager.begin();
    GlobalState workingState = currentState;

    try {
        // Step 1: Transfer all resources across all types
        for (AgentId agent : result.getAllocatedAgents()) {
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                long quantity = result.getAllocation(agent, type);
                if (quantity > 0) {
                    workingState = workingState.getResourceLedger()
                        .transfer(POOL_ID, agent, type, quantity)
                        .updateIn(workingState);
                }
            }
        }

        // Step 2: Burn currency from all agents
        for (Map.Entry<AgentId, BigDecimal> entry :
                result.getCurrencyBurned().entrySet()) {
            workingState = workingState.getResourceLedger()
                .burnCurrency(entry.getKey(), entry.getValue())
                .updateIn(workingState);
        }

        // Step 3: Set lease expirations for all (agent, resource) pairs
        Instant now = Instant.now();
        for (AgentId agent : result.getAllocatedAgents()) {
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                if (result.getAllocation(agent, type) > 0) {
                    Duration leaseDuration = getLeaseDuration(type);
                    workingState = workingState.getResourceLedger()
                        .setLeaseExpiration(agent, type,
now.plus(leaseDuration))
                        .updateIn(workingState);
```

```
                }
            }
        }

        // Step 4: Update agent states with new resources
        for (AgentId agent : result.getAllocatedAgents()) {
            ResourceBundle newBundle = ResourceBundle.empty();
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                long quantity = result.getAllocation(agent, type);
                if (quantity > 0) {
                    newBundle = newBundle.with(type, quantity);
                }
            }

            ImmutableAgentState agentState =
workingState.getAgentState(agent);
            ImmutableAgentState updatedState = agentState.withResources(
                agentState.getCurrentResources().merge(newBundle)
            );
            workingState = workingState.withAgentState(agent,
updatedState);
        }

        // Validate before commit (all invariants must hold)
        ValidationResult validation =
safetyMonitor.validateState(workingState);
        if (!validation.isValid()) {
            throw new TransactionException(
                "Safety validation failed: " + validation.getViolations()
            );
        }

        // Commit transaction atomically
        GlobalState committedState = transactionManager.commit(txn,
workingState);

        logger.info("Joint allocation enforced: {} agents, {} resource
types, " +
                   "total problem size {}",
```

```
                    result.getAllocatedAgents().size(),
                    result.getAllocatedResourceTypes().size(),
                    result.getAllocatedAgents().size() *
                        result.getAllocatedResourceTypes().size());

            return committedState;

        } catch (Exception e) {
            // Any failure triggers rollback - no partial allocations
            transactionManager.rollback(txn, currentState);
            logger.error("Joint allocation enforcement failed, rolled back",
e);
            throw new TransactionException(
                "Failed to enforce joint allocation atomically", e
            );
        }
}
```

Why This Atomicity Matters:

The detailed enforcement process makes explicit the many operations involved in a joint allocation. Without atomicity (Property 6.3), failure partway through would leave the system in an inconsistent state:

Bad scenario without atomicity:

```
Joint allocation: {A₁, A₂, A₃} × {compute, storage, dataset}

Step 1: Allocate compute to A₁ ✓
Step 2: Allocate compute to A₂ ✓
Step 3: Allocate compute to A₃ ✓
Step 4: Allocate storage to A₁ ✓
Step 5: Allocate storage to A₂ ✗ ERROR: insufficient storage (bug in
arbitrator)
Step 6: [Not reached] Allocate storage to A₃
Step 7: [Not reached] Allocate dataset...
Step 8: [Not reached] Burn currency...

Result:
- A₁ has compute (good) but not storage yet (inconsistent)
```

```
- A₂ has compute (good) but can't get storage (broken)
- A₃ has compute (good) but won't get storage/dataset (broken)
- No currency burned yet (agents got resources for free!)
- System is in partial-allocation limbo
```

With atomicity via TransactionManager:

```
Step 5's error triggers rollback of Steps 1-4.
All agents return to their pre-arbitration state.
No resources transferred, no currency burned.
Contention remains, will be re-detected next cycle.
Platform can investigate why arbitrator produced infeasible allocation (bug).
System remains consistent throughout.
```

The rollback capability—enabled by immutable state where we can simply discard the working state and retain the original—is what makes atomic joint allocations practical. We don't need complex compensation logic or undo operations. We just don't commit the new state if validation fails.

Release Process for Multi-Resource Bundles:

When releaseResources() is called with a bundle containing multiple resource types (common after joint allocations):

```
public GlobalState releaseResources(
        AgentId agentId,
        ResourceBundle resources,
        GlobalState currentState) {

    // Validate agent owns all resources being released
    for (Map.Entry<ResourceType, Long> entry : resources.entrySet()) {
        long held = currentState.getResourceLedger()
            .getResourceQuantity(agentId, entry.getKey());
        if (held < entry.getValue()) {
            throw new InvalidReleaseException(
                "Agent " + agentId + " doesn't hold " + entry.getValue() +
                " units of " + entry.getKey() + " (has " + held + ")"
            );
        }
```

```
    }

    // Calculate earnings per resource type
    Instant now = Instant.now();
    Map<ResourceType, Instant> leaseExpiries =
        currentState.getResourceLedger().getLeaseExpirations(agentId);

    Map<ResourceType, BigDecimal> earningsPerType = new HashMap<>();
    BigDecimal totalEarnings = BigDecimal.ZERO;

    for (Map.Entry<ResourceType, Long> entry : resources.entrySet()) {
        ResourceType type = entry.getKey();
        long quantity = entry.getValue();

        Instant expiry = leaseExpiries.get(type);
        long timeRemainingSeconds = Duration.between(now,
expiry).getSeconds();

        BigDecimal multiplier = priorityWeightCalculator
            .calculateDemandMultiplier(type, currentState);

        BigDecimal earnings = BigDecimal.valueOf(quantity)
            .multiply(BigDecimal.valueOf(timeRemainingSeconds))
            .multiply(multiplier);

        earningsPerType.put(type, earnings);
        totalEarnings = totalEarnings.add(earnings);
    }

    // Build new state with releases and earnings
    GlobalState newState = currentState;

    // Transfer resources back to pool (all types)
    for (Map.Entry<ResourceType, Long> entry : resources.entrySet()) {
        newState = newState.getResourceLedger()
            .transfer(agentId, POOL_ID, entry.getKey(), entry.getValue())
            .updateIn(newState);
    }
```

```
    // Mint total earnings
    newState = newState.getResourceLedger()
        .mintCurrency(agentId, totalEarnings)
        .updateIn(newState);

    // Clear lease expirations for released resources
    for (ResourceType type : resources.keySet()) {
        newState = newState.getResourceLedger()
            .clearLeaseExpiration(agentId, type)
            .updateIn(newState);
    }

    // Publish release event with earnings breakdown
    eventBus.publish(new ResourceReleaseEvent(
        agentId,
        resources,
        leaseExpiries,
        earningsPerType,
        totalEarnings,
        "voluntary_release"  // or other reason
    ));

    return newState;
}
```

The multi-resource release process maintains the same earning formula per resource type (Quantity × TimeRemaining × DemandMultiplier) but applies it independently to each type and sums the results. An agent releasing {compute:50, storage:30, dataset:2} after a joint allocation earns:

```
earnings_compute = 50 × timeRemaining_compute × multiplier_compute
earnings_storage = 30 × timeRemaining_storage × multiplier_storage
earnings_dataset = 2 × timeRemaining_dataset × multiplier_dataset
total = earnings_compute + earnings_storage + earnings_dataset
```

This independence across resource types keeps the earning formula simple and predictable. Phase 2 might add complementarity bonuses (extra earnings for releasing resources that were

allocated together in a joint optimization, since releasing them together maintains the beneficial trade structure), but Milestone 1's independence is sufficient and easier to reason about.

Conservation Validation Across All Types:

Before committing any state transition involving resource transfers, the SafetyMonitor (called by TransactionManager) validates conservation for every resource type:

```
public ValidationResult validateState(GlobalState state) {
    List<String> violations = new ArrayList<>();

    // Check resource conservation for all types
    for (ResourceType type : state.getResourceTypes()) {
        long total = 0;

        // Sum across all agents
        for (AgentId agent : state.getAllAgents()) {
            total += state.getResourceLedger()
                .getResourceQuantity(agent, type);
        }

        // Add pool quantity
        total += state.getResourceLedger().getPoolQuantity(type);

        long expected = initialQuantities.get(type);
        if (total != expected) {
            violations.add(String.format(
                "Conservation violated for %s: total=%d, expected=%d",
                type, total, expected
            ));
        }
    }

    // [Additional invariant checks...]

    return violations.isEmpty()
        ? ValidationResult.valid()
        : ValidationResult.invalid(violations);
}
```

For joint allocations touching 5 resource types and 10 agents, this validation checks 5 conservation equations. If any fails, the entire transaction rolls back. This defense-in-depth catches bugs in arbitration logic (allocated more than exists) or enforcement logic (incorrectly transferred resources) before they corrupt the system state.

7.2 Priority Economy: Earning and Spending Dynamics

The Priority Economy creates a closed-loop system where currency flows in a cycle: agents earn by releasing resources → accumulate balances → burn for priority → receive better allocations → eventually release → earn again. The currency itself has no value outside the system—it's purely a priority signaling mechanism, not a medium of exchange. The Priority Economy operates identically across all resource types, with earnings and spending affecting joint allocations uniformly.

Earning Formula Deep Dive:

The earning formula's application to multi-resource releases illustrates its closed-loop dynamics clearly:

```
For each resource type j being released:
  earnings_j = quantity_j × TimeRemaining_j × DemandMultiplier_j

Total earnings = Σ_j earnings_j
```

Quantity component: Linear scaling—releasing 100 units earns 10× releasing 10 units. This proportionality makes hoarding expensive (opportunity cost of foregone earnings is high). For agents who received joint allocations across multiple resource types, they can release types independently (as tasks using specific types complete) or together (when overall task completes), earning currency in either case.

TimeRemaining component: Incentivizes early release. If lease duration is 10 timesteps and agent releases after 2 timesteps used, TimeRemaining = 8. If they wait until timestep 9, TimeRemaining = 1. The 8× multiplier from early release strongly incentivizes returning unused capacity promptly. For joint allocations with different lease durations per resource type (compute might have 20-timestep leases, lab_time might have 5-timestep leases), the TimeRemaining is calculated independently per type, creating different earning rates that reflect resource-specific policies.

The platform could use alternative time functions in Phase 2:

- **Linear** (current): earnings $\propto$ TimeRemaining
- **Exponential**: earnings $\propto$ exp(TimeRemaining/LeaseDuration), even stronger early-release incentive
- **Step function**: bonus for releasing before halfway point, no bonus after

Milestone 1 uses linear for simplicity. Phase 2 can experiment with nonlinear time incentives and measure their impact on release behavior and system liquidity.

DemandMultiplier component: Reflects scarcity per resource type. When demand >> supply (multiplier = 2.0-3.0), releasing scarce resources earns premium. When supply >> demand (multiplier = 0.5-0.8), releasing abundant resources earns less. This creates natural equilibration—agents have incentive to release scarce resources (high earnings) and hold abundant resources (low earnings), which tends to balance supply toward demand.

For joint allocations, different resource types have different multipliers simultaneously. An agent releasing {compute:50, storage:30} when compute is scarce (multiplier=2.5) and storage is abundant (multiplier=0.8) earns much more from compute than storage. This differential creates incentives to release high-multiplier resources first if possible, further accelerating equilibration for contested types.

Mathematically:

```
DemandMultiplier_j = TotalDemand_j / AvailableSupply_j


where:
  TotalDemand_j = Σ_i (agent i's current ideal request for resource j)
  AvailableSupply_j = current pool quantity of resource j
```

Edge case handling:

- If AvailableSupply = 0 (pool exhausted): multiplier = FIXED_HIGH (e.g., 10.0) to strongly incentivize releases into empty pool without causing numerical instability from division by zero
- If TotalDemand = 0 (no one wants resource): multiplier = 0.5 to avoid zero earnings but reflect low scarcity

Multi-Resource Release Example:

Agent releases resources from joint allocation after completing a task:

```
Released bundle: {compute:50, storage:30, dataset:2}
Lease durations: {compute:20 timesteps, storage:20, dataset:20}
Time used: 5 timesteps
TimeRemaining: compute:15, storage:15, dataset:15 (all released together)

Current demand multipliers:
  compute: 2.5 (high contention - many agents want compute)
  storage: 1.2 (moderate contention)
  dataset: 0.8 (low contention - dataset is abundant)
```

```
Earnings breakdown:
  compute: 50 × 15 × 2.5 = 1,875 currency
  storage: 30 × 15 × 1.2 = 540 currency
  dataset: 2 × 15 × 0.8 = 24 currency

Total earnings: 2,439 currency units
```

The agent earns substantially more from releasing compute (scarce) than dataset (abundant), even though they released more dataset than they would have if considering earnings alone. This is fine—the agent needed the dataset for their task (high preference weight w_dataset), and the earning differential doesn't change the past. But for future acquisitions, the agent might consider: "dataset is cheap to hold (low earning multiplier), while compute is expensive (high multiplier). Maybe I should release compute quickly and hold dataset longer."

This forward-looking consideration is part of the emergent strategic behavior the Priority Economy incentivizes. Agents learn that releasing scarce resources is lucrative and adjust their acquisition-release patterns accordingly.

Spending: Currency as Priority Signal

Agents spend currency not through payment (transferring to other agents or platform) but through **burning** (permanent destruction). When arbitration occurs for a contention (single-resource or joint):

1. **Agent called:** Platform calls `agent.decidePriorityBudget(state, need)`
2. **Agent decides:** Agent examines balance, need urgency across all resource types in the request, expected future needs, returns amount to burn
3. **Platform validates:** Checks balance ≥ amount (or balance ≥ -100 if allowing bounded debt)
4. **Weight calculated:** `priorityWeight = BaseWeight + burnedAmount = 10.0 + burnedAmount`
5. **Arbitration:** Arbitrator uses priority weight in objective: `maximize Σ_i priorityWeight_i · log(Φ_i)` where Φ_i sums utility across all contested resource types
6. **Burning:** After arbitration, ResourceManager calls `ledger.burnCurrency(agentId, burnedAmount)` within the atomic transaction, destroying the currency

The destroyed currency doesn't go anywhere—it ceases to exist. This burn-not-pay model prevents wealth accumulation cycles where early winners accumulate currency that enables future wins, creating runaway inequality. By destroying currency when spent, we reset everyone toward baseline over time (unless they earn more).

Strategic Considerations for Joint Contentions:

When an agent knows their request will likely be grouped into a joint contention (because they're requesting multiple resource types that many agents want), the strategic calculus for decidePriorityBudget() becomes more interesting:

Single-resource contention: Burning 50 currency gives priority for one resource type. Expected return: better allocation for that specific resource.

Joint contention: Burning 50 currency gives priority across ALL resource types in the joint optimization. Expected return: better allocations across compute, storage, dataset, etc. simultaneously.

The leverage from burning currency in joint contentions is higher because the priority weight affects the agent's total utility $\Phi_i = \Sigma_j w_{ij} \cdot a_{ij}$, which spans multiple resource types. An agent with w_compute=0.5, w_storage=0.3, w_dataset=0.2 benefits from increased allocations across all three types when their priority weight is elevated.

This creates an interesting strategic insight: **joint contentions make priority spending more effective**. An agent who would burn 20 currency for single-resource contention might choose to burn 40 currency for joint contention because the increased priority affects more dimensions. Over time, agents might learn to be more aggressive with currency spending when they anticipate joint grouping (many resource types contested with overlapping agents) and more conservative when they anticipate single-resource arbitrations (isolated contentions).

Three-Tier Outcome Space:

The interaction between BaseWeight, burned currency, and logarithmic objective creates a continuous spectrum of outcomes, but conceptually there are three tiers:

Tier 1: Urgent agents with currency (high priority weight c=100-500):
- Receive allocations quickly and generously across contested resource types
- Get close to ideal requests (80-95% of ideal) even in severe contention
- Can sustain high spending because they earn back through releases
- Example: Agent with critical deadline burns 200 currency, receives {compute:90/100, storage:45/50}

Tier 2: Normal agents with moderate currency (medium priority weight c=20-100):
- Receive allocations at moderate speed and quantity
- Get reasonable partial allocations (50-70% of ideal) in contention
- Balance earning and spending, maintaining stable currency reserves
- Example: Agent burns 30 currency, receives {compute:60/100, storage:30/50}

Tier 3: Low-resource agents with zero currency (baseline priority weight c=10):
- Still receive allocations (logarithmic barrier ensures this), just slower and smaller
- Get minimal but sufficient partial allocations (25-40% of ideal) in severe contention
- Must rely on earning through releases before they can spend for priority
- Example: Agent burns 0 currency, receives {compute:30/100, storage:15/50} - enough to function and earn back

The key insight is that **Tier 3 never drops to zero**—the BaseWeight=10.0 ensures participation, and the logarithmic barrier prevents wealthy agents in Tiers 1-2 from excluding Tier 3 agents completely. This is the "constitutional guarantee" of the mechanism: a participation floor that no amount of currency burning can override.

For joint allocations, this three-tier structure applies across all resource types simultaneously. A Tier 3 agent receives below-average allocations for compute, storage, and dataset, but receives *something* for all three, enabling continued operation and earning opportunities.

Equilibrium Currency Balances:

Over time, agents' balances reach equilibria where earning rates balance spending rates:

```
Equilibrium balance C̄i satisfies:
  Earning rate = Release frequency × Average earnings per release
  Spending rate = Arbitration frequency × Average burn per arbitration

At equilibrium: Earning rate ≈ Spending rate

Agents with C̄i > 1000 (high balance):
  - Earn frequently (release often, possibly multi-resource releases
earning more) or burn sparingly
  - Can afford aggressive burning when needed
  - In good "financial health"

Agents with C̄i near 0 (low balance):
  - Spend faster than earning or release infrequently
  - Must burn cautiously, saving for critical needs
  - In "financial stress"

Agents with C̄i < 0 (in debt):
  - Temporarily overspent, must earn back through releases
  - Debt limit (-100) prevents runaway negative balances
```

The self-regulating property of these equilibria is what makes the Priority Economy stable. Agents who overspend find themselves unable to continue spending at high rates (balance depleted), forcing moderation. Agents who underspend accumulate balances, eventually finding opportunities where spending makes sense. No central bank or policy intervention needed—the dynamics are emergent from the incentive structure.

For agents participating in joint contentions frequently (because they request resource types that group often), equilibrium balances might be higher because the leverage from spending is greater. Conversely, agents with niche resource interests (rarely involved in joint contentions) might maintain lower equilibria because spending is less effective for them.

7.3 Semantic Divergence in Practice

Theorem 4 and the refined Theorem 4.1 establish bounds on how observable optimization (what the platform does) diverges from semantic optimization (what agents truly care about). While the theoretical framework is unchanged, the architecture can support lightweight monitoring of semantic divergence to guide configuration quality improvement.

Observable vs. Semantic Utility Gap:

The platform optimizes observable utility Φ_obs(A) = Σ_j w_ij · a_ij across all resource types using declared preference weights. But agents' true semantic utility Φ_sem(A) includes hidden components: long-term consequences of resource usage, domain-specific meanings, emergent effects from multi-resource allocations. The gap Φ_hidden = Φ_sem - Φ_obs is what the platform cannot see and therefore cannot directly optimize.

Theorem 4 bounds this gap: |Φ_sem(A*_obs) - Φ_sem(A*_sem)| ≤ 2·||Φ_hidden||_∞. Theorem 4.1 refines this when hidden utility correlates with observable utility: the bound becomes (1-|ρ|)·2·||Φ_hidden||_∞ where ρ is the correlation coefficient.

Architectural Support for Measuring Divergence:

The platform can estimate semantic divergence through post-hoc feedback:

```
public interface SemanticFeedbackCollector {
    /**
     * After allocation, ask agent to rate how well it satisfied true
goals.
     * Optional - agents can decline to provide feedback.
     *
     * @param agentId Agent to query
     * @param allocation Allocation agent received
     * @param observableUtility Platform's computed utility
     * @return Agent's true semantic utility, or null if no feedback
     */
    @Nullable
    Double collectSemanticUtility(
        AgentId agentId,
        ResourceAllocation allocation,
        double observableUtility
    );

    /**
     * Estimate ||Φ_hidden||_∞ for agent based on historical feedback.
     * Used to assess configuration quality.
     */
    double estimateHiddenUtilityMagnitude(AgentId agentId);

    /**
```

```
     * Estimate correlation ρ between hidden and observable utility.
     * Used to apply Theorem 4.1's refined bound.
     */
    double estimateCorrelation(AgentId agentId);
}
```

By collecting semantic feedback over time, the platform builds estimates of ||Φ_hidden||_∞ and ρ per agent. These estimates enable:

Configuration quality metrics: Display to agent creators showing "your agent's hidden utility magnitude is estimated at 15 units, suggesting 30-unit potential divergence from optimal allocations. Consider adding observable features to reduce this."

Refined bound application: Use Theorem 4.1's correlation-based bound (1-|ρ|)·2·||Φ_hidden||_∞ for more accurate divergence estimates when correlation data is available.

Trend analysis: Track whether divergence is increasing (agent's needs evolving beyond their configuration's expressiveness) or decreasing (agent refining configuration based on experience).

For Milestone 1-2, semantic feedback collection is optional and manual (researchers ask agents about satisfaction in test scenarios). Phase 2 might automate this by having agents implement an optional `reportSemanticUtility()` method that the platform calls periodically, though this requires trusting agents to report honestly (which they have incentive to do if reporting helps improve future allocations).

Proposition 4.1's Strategic Implications:

The theory document's Proposition 4.1 argues heuristically that declaring comprehensive observable preferences (minimizing ||Φ_hidden||_∞) is approximately optimal for agents because the bounded divergence theorem limits gains from strategic misrepresentation. Architecturally, we support this through:

Configuration templates: Provide example configurations with comprehensive observable features across multiple resource types, demonstrating best practices for minimizing hidden utility.

Validation warnings: During configuration submission, warn if preference weights seem incomplete (agent requests many resource types but only specifies weights for a few, suggesting hidden preferences).

Feedback loops: After agents observe outcomes from several allocations, suggest configuration refinements based on whether allocations met true goals.

These lightweight mechanisms create gentle pressure toward high-quality configurations without requiring complex incentive schemes. The architecture doesn't enforce comprehensive

configuration (agents can submit sparse preferences if they choose), but it makes comprehensive configuration advantageous through better allocation outcomes.

7.4 Currency Balance Tracking and Debt Management

Currency balance management remains straightforward but is worth reinforcing given its importance to the Priority Economy's stability:

Balance Updates (Minting):

When resources are released, PriorityWeightCalculator computes earnings (summed across all released types), and ResourceLedger mints currency:

```
public ImmutableResourceLedger mintCurrency(AgentId agent, BigDecimal
amount) {
    BigDecimal currentBalance = currencyBalances.get(agent);
    BigDecimal newBalance = currentBalance.add(amount);

    // Update balance map (immutable update via persistent data structure)
    ImmutableMap<AgentId, BigDecimal> newBalances =
        currencyBalances.put(agent, newBalance);

    // Track cumulative minting for currency accounting invariant
    BigDecimal newCumulativeMinting = cumulativeMinting.add(amount);

    return new ImmutableResourceLedger(
        resourceHoldings,
        newBalances,
        leaseExpirations,
        newCumulativeMinting,
        cumulativeBurning
    );
}
```

Balance Updates (Burning):

When arbitration occurs, ResourceLedger burns currency within the atomic transaction:

```
public ImmutableResourceLedger burnCurrency(AgentId agent, BigDecimal
amount) {
```

```
    BigDecimal currentBalance = currencyBalances.get(agent);
    BigDecimal newBalance = currentBalance.subtract(amount);

    // Validate debt limit
    if (newBalance.compareTo(BigDecimal.valueOf(-100)) < 0) {
        throw new InsufficientCurrencyException(
            "Burning " + amount + " would exceed debt limit for agent " +
agent
        );
    }

    ImmutableMap<AgentId, BigDecimal> newBalances =
        currencyBalances.put(agent, newBalance);

    // Track cumulative burning for currency accounting invariant
    BigDecimal newCumulativeBurning = cumulativeBurning.add(amount);

    return new ImmutableResourceLedger(
        resourceHoldings,
        newBalances,
        leaseExpirations,
        cumulativeMinting,
        newCumulativeBurning
    );
}
```

Bounded Debt Mechanism:

The debt limit (-100 minimum balance) provides flexibility while preventing instability:

Scenario: Agent has balance=90, wants to burn 100 currency for urgent joint contention

- Without debt: Cannot burn 100, must burn only 90, gets slightly lower priority
- With bounded debt: Can burn 100, going to balance=-10, gets full priority desired
- Consequence: Must earn back at least 10 currency before burning again, creating pressure to release resources promptly

The debt limit is conservative (only -100 compared to initial endowment of 1000) to prevent agents from accumulating large debts that might never be repaid. For Milestone 1-2, we'll monitor whether agents actually use debt (how often do they go negative? how long do they stay negative? do they reliably recover?). Phase 2 might adjust the limit based on empirical data or make it agent-specific (higher-reputation agents get higher debt limits).

Currency Accounting Invariant:

The ResourceLedger tracks cumulative minting and burning to enable validation of currency accounting (Invariant I4 from Section 9.1):

```
Total currency in existence = Σ_i balance_i
Expected total = InitialEndowments + CumulativeMinting - CumulativeBurning

Invariant: Total currency = Expected total (accounting consistency)
```

This invariant catches currency leaks (minting without recording, burning without decrementing) or duplication (incorrectly minting twice, incorrectly failing to burn). For a closed-loop economy where currency supply affects equilibrium dynamics, accurate accounting is essential.

---

8. Arbitration Engine: Proportional Fairness via Convex Optimization

The arbitration engine is where theoretical mechanism design meets practical implementation. This section provides detailed specifications for the ProportionalFairnessArbitrator, including the exponential cone formulation for joint problems, integer constraint handling, and error handling strategies for large-scale joint contentions.

8.1 Exponential Cone Formulation for Joint Contentions

The Weighted Proportional Fairness objective—maximize Σ_i c_i · log(Φ_i(A))—is most naturally solved using exponential cone programming, regardless of whether we're optimizing over one resource type or many. The formulation extends cleanly from single-resource to joint contentions by expanding the utility function Φ_i(A) to sum across all contested resource types rather than just one.

Mathematical Reformulation for Joint Problems:

Original problem for n agents and m resource types:

```
maximize: Σ_i c_i · log(Φ_i(A))

where:
  Φ_i(A) = Σ_jϵcontested w_ij · a_ij (sum across contested resource types)

subject to:
```

```
  Σ_i a_ij ≤ Q_j (resource limit per type j)
  a_ij ≥ min_ij (minimums per agent i, resource j)
  a_ij ≤ ideal_ij (maximums per agent i, resource j)
  Φ_i(A) > 0 (positive utility)
```

Exponential cone reformulation:

```
maximize: Σ_i c_i · t_i

subject to:
  (t_i, Φ_i(A), 1) ∈ K_exp  for all i
  Φ_i(A) = Σ_j∈contested w_ij · a_ij (expands across all m contested types)
  [linear constraints on a_ij for all j]

where K_exp = {(z, y, 1) : y > 0, y·exp(z/y) ≥ 1}
```

The key difference from single-resource formulation is that each agent's utility $\Phi_i(A)$ now sums over m resource types (all types in the joint contention) rather than just one. The exponential cone constraint $(t_i, \Phi_i(A), 1) \in K_{exp}$ still enforces $t_i = \log(\Phi_i(A))$ at optimality, but now Φ_i represents the agent's total utility across the full resource bundle.

Problem Dimensions:

For joint contention with n agents and m resource types:

- **Allocation variables:** n × m (one per agent-resource pair)
- **Auxiliary variables:** n (one t_i per agent for log utility)
- **Total variables:** n×m + n = n(m+1)
- **Exponential cone constraints:** n (one per agent)
- **Linear constraints:** m (resource limits) + 2nm (mins and maxes per agent-resource)

For Milestone 1 worst case (n=20, m=10): 20×10 + 20 = 220 variables, 20 exponential cones, 10 + 400 = 410 linear constraints. Clarabel handles this in 200-500ms as shown in Appendix B's complexity analysis.

Why Joint Optimization Achieves Global Pareto Optimality:

The mathematical foundation is that we're optimizing over the full allocation matrix $A = [a_{ij}]$ where i ranges over all agents in the joint contention and j ranges over all resource types in the joint contention. By Theorem 1, the solution maximizing $\Sigma_i c_i \cdot \log(\Phi_i(A))$ is Pareto-optimal within the scope of the optimization. Aggressive grouping sets the scope to include all contested resources and all agents requesting any contested resource, ensuring global Pareto optimality (Property 2.3).

The constructive interpretation adjuncted to Theorem 1's proof shows that any feasible direction that improves one agent's utility must necessarily decrease another agent's utility—this is

precisely the definition of being on the Pareto frontier. For joint problems, this means we cannot find any reallocation across any subset of the contested resources that would make some agent better off without hurting another.

Solver Integration: Clarabel for Joint Problems:

Clarabel's exponential cone methods scale naturally to joint problems. The solver doesn't distinguish between "single-resource optimization with 20 agents" and "joint optimization with 20 agents × 10 resource types"—both are exponential cone programs with appropriate dimensions. The solver's polynomial-time guarantee O((nm)^2.5) applies uniformly.

Building Joint Problem for Clarabel:

```
private ClarabelProblem buildJointProblem(
        Contention contention,
        Map<AgentId, Double> priorityWeights,
        Map<AgentId, PreferenceFunction> preferences) {

    List<AgentId> agents = new ArrayList<>(contention.getInvolvedAgents());
    List<ResourceType> resourceTypes =
        new ArrayList<>(contention.getResourceTypes());

    int n = agents.size();
    int m = resourceTypes.size();
    int numAllocVars = n * m;  // allocation matrix variables
    int numAuxVars = n;         // auxiliary t_i variables
    int numVars = numAllocVars + numAuxVars;

    ClarabelProblem.Builder builder = ClarabelProblem.builder(numVars);

    // Set objective: maximize Σ_i c_i · t_i
    // Auxiliary variables t_i are at indices [n×m, n×m+n)
    for (int i = 0; i < n; i++) {
        AgentId agent = agents.get(i);
        double c_i = priorityWeights.get(agent);
        builder.setObjectiveCoefficient(numAllocVars + i, c_i);
    }

    // Add exponential cone constraints: (t_i, Φ_i(A), 1) in K_exp
    // where Φ_i(A) = Σ_j w_ij · a_ij across all contested resource types
```

```
    for (int i = 0; i < n; i++) {
        AgentId agent = agents.get(i);
        PreferenceFunction pref = preferences.get(agent);

        // Collect all allocation variables for agent i across all types
        int[] allocVarIndices = new int[m];
        double[] prefWeights = new double[m];

        for (int j = 0; j < m; j++) {
            ResourceType type = resourceTypes.get(j);
            allocVarIndices[j] = i * m + j;  // row-major indexing
            prefWeights[j] = pref.getWeight(type);
        }

        // Add cone constraint: (t_i, Σ_j w_ij·a_ij, 1) in K_exp
        builder.addExponentialCone(
            numAllocVars + i,  // t_i variable index
            allocVarIndices,    // allocation variables for agent i
            prefWeights         // preference weights
        );
    }

    // Add resource limit constraints: Σ_i a_ij ≤ Q_j for each type j
    for (int j = 0; j < m; j++) {
        ResourceType type = resourceTypes.get(j);
        long Q_j = contention.getAvailableSupply(type);

        // Collect variables: a_0j, a_1j, ..., a_(n-1)j (column j of
matrix)
        int[] columnIndices = new int[n];
        double[] coefficients = new double[n];
        for (int i = 0; i < n; i++) {
            columnIndices[i] = i * m + j;
            coefficients[i] = 1.0;
        }

        builder.addLinearConstraint(
            columnIndices,
```

```
            coefficients,
            Q_j,
            ConstraintType.LEQ
        );
    }

    // Add minimum constraints: a_ij ≥ min_ij for all (i,j)
    for (int i = 0; i < n; i++) {
        AgentId agent = agents.get(i);
        ResourceRequest req = contention.getRequestForAgent(agent);

        for (int j = 0; j < m; j++) {
            ResourceType type = resourceTypes.get(j);
            long min_ij = req.getMinimum(type);

            if (min_ij > 0) {
                builder.addLinearConstraint(
                    new int[]{i * m + j},
                    new double[]{1.0},
                    min_ij,
                    ConstraintType.GEQ
                );
            }
        }
    }

    // Add maximum constraints: a_ij ≤ ideal_ij for all (i,j)
    for (int i = 0; i < n; i++) {
        AgentId agent = agents.get(i);
        ResourceRequest req = contention.getRequestForAgent(agent);

        for (int j = 0; j < m; j++) {
            ResourceType type = resourceTypes.get(j);
            long ideal_ij = req.getIdeal(type);

            builder.addLinearConstraint(
                new int[]{i * m + j},
```

```
                new double[]{1.0},
                ideal_ij,
                ConstraintType.LEQ
            );
        }
    }

    return builder.build();
}
```

This formulation creates a problem with n(m+1) variables and m + 2nm linear constraints plus n exponential cone constraints. The structure is sparse (each variable appears in only a few constraints), which Clarabel exploits for efficient matrix operations.

Key Architectural Insight:

The formulation code is identical in structure whether m=1 (single-resource) or m=10 (joint). The arbitrator doesn't have separate code paths for single vs. joint—it just scales the problem dimensions. This uniformity simplifies implementation and testing: we validate the joint formulation thoroughly, and single-resource arbitrations become a special case (m=1).

Interface Design:

The arbitrator interface remains clean and doesn't expose whether contentions are joint or single:

```
public interface ProportionalFairnessArbitrator {
    /**
     * Compute globally Pareto-optimal allocation for contention.
     * Contention may be single-resource (m=1) or joint (m>1).
     * Optimization considers full allocation matrix for all types.
     *
     * @param contention Resource conflict(s) to resolve
     * @param state Current platform state (for preferences, currency)
     * @return Allocation across all resource types in contention
     * @throws ArbitrationException if optimization fails or infeasible
     */
    AllocationResult arbitrate(
        Contention contention,
        ImmutableGlobalState state
    ) throws ArbitrationException;
```

```
    /**
     * Check if contention is likely to produce large optimization problem.
     * Used for monitoring and potential fallback decisions in Phase 2.
     *
     * @param contention Contention to assess
     * @return Problem size (agents × resource types)
     */
    default int estimateProblemSize(Contention contention) {
        return contention.getInvolvedAgents().size() *
               contention.getResourceTypes().size();
    }
}
```

8.2 Integer Constraint Handling for Joint Allocations

The convex formulation assumes continuous quantities ($a_{ij} \in \mathbb{R}^+$), but Milestone 1 requires integer allocations ($a_{ij} \in \mathbb{N}$) for discrete resources. The strategy remains solve-continuous-then-round, but with additional considerations for joint problems where rounding errors accumulate across multiple resource types.

Rationale:

Mixed-integer convex programming (MICP) can have significantly higher complexity than pure convex programming. While convex programs are solvable in polynomial time, MICP might require branch-and-bound (exponentially many branches in worst case). By solving continuous relaxation and rounding, we maintain polynomial-time guarantee (Theorem 2) while accepting near-optimality for integer solutions.

For joint allocations with n=20 agents and m=10 resource types, we have 200 allocation variables. Rounding each independently could accumulate error: if each rounds by ±0.5, total error could be ±100 units. However, errors partially cancel (some round up, others down), and the total error across each resource type is typically 5-15 units for entire allocation, acceptable when pool is 1000+ units.

Rounding Strategy for Joint Allocations:

Phase 1: Solve continuous relaxation

```
1. Formulate exponential cone problem with a_ij ∈ ℝ⁺
2. Call Clarabel.solve(problem)
3. Receive optimal continuous solution a*_ij ∈ ℝ⁺ for all (i,j)
```

Phase 2: Round to nearest integer

```
4. For each a_ij, compute â_ij = round(a*_ij) using standard rounding (0.5
rounds up)
```

Phase 3: Validate feasibility across all resource types

```
5. Check resource limits per type: Σ_i â_ij ≤ Q_j for all j

6. Check minimums per agent-resource: â_ij ≥ min_ij for all i,j where
min_ij > 0

7. If ALL constraints satisfied across ALL types, accept â as final
allocation

8. If ANY constraints violated for ANY type, proceed to Phase 4
```

Phase 4: Iterative feasibility restoration (if needed)

```
9. Identify violated constraints (which resource types exceed limits? which
minimums violated?)

10. For resource limit violations:

    - For each resource type j where Σ_i â_ij > Q_j:

      - Compute excess: excess_j = Σ_i â_ij - Q_j

      - Reduce allocations for agents with most slack (farthest from
minimums)

      - Distribute reduction proportionally across agents to minimize
utility loss

11. For minimum violations:

    - For each (i,j) where â_ij < min_ij:

      - Increase to exactly min_ij

      - Reduce other agents' allocations of type j proportionally to
compensate

12. Re-check feasibility across ALL types

13. Repeat steps 10-12 up to MAX_ITERATIONS (e.g., 10)

14. If still infeasible after MAX_ITERATIONS, signal failure
```

Phase 5: Fallback (if iterative restoration fails)

```
15. Option A: Reject entire joint allocation, signal infeasibility to all
agents

16. Option B: Use floor(a*_ij) instead of round (always rounds down, more
conservative)

17. Option C: Solve with increased tolerance (relax constraints slightly,
accept approximate feasibility)
```

Example Feasibility Restoration for Joint Problem:

Suppose continuous solution for joint contention with agents {A_1, A_2} and resources {compute, storage} gives:

```
Matrix:
        compute  storage
A1:     47.6     52.7
A2:     52.4     47.3

Resource limits: Q_compute = 100, Q_storage = 100
```

Rounding:

```
A1: compute:48, storage:53
A2: compute:52, storage:47

Check totals:
  compute: 48 + 52 = 100 ✓ (exactly at limit)
  storage: 53 + 47 = 100 ✓ (exactly at limit)

Feasible! Accept rounded solution.
```

Suppose instead a third agent and dataset were included with marginal rounding issues:

```
Matrix after rounding:
        compute  storage  dataset
A1:     48       53       7
A2:     52       47       3
A3:     1        1        1

Totals:
  compute: 48+52+1 = 101 > 100 ✗ (exceeds by 1)
  storage: 53+47+1 = 101 > 100 ✗ (exceeds by 1)
  dataset: 7+3+1 = 11 > 10 ✗ (exceeds by 1)

All three types need reduction. Restore feasibility:
```

```
Iteration 1:
  - Compute: Reduce A₁ by 1 (has most slack: 48 vs minimum, assume min=40)
    → A₁: compute:47
  - Storage: Reduce A₁ by 1 (has most slack: 53 vs minimum, assume min=45)
    → A₁: storage:52
  - Dataset: Reduce A₁ by 1 (has most slack: 7 vs minimum, assume min=5)
    → A₁: dataset:6

Check totals: All now exactly 100, 100, 10 respectively ✓ Feasible!
```

The restoration algorithm prioritizes reducing allocations from agents with the most slack (farthest above their minimums), minimizing the utility loss from rounding adjustments. For most joint problems (>90%), simple rounding produces feasible solutions. Edge cases require 1-3 iterations of restoration. Rare cases (<1%) might require fallback.

Why Not Full Branch-and-Bound for Joint Problems:

Full branch-and-bound for mixed-integer convex programs with 200 variables could have exponential complexity. For a safety-critical platform, maintaining polynomial-time guarantee (no exponential worst-case) is more important than achieving exact integer optimality. The continuous-then-round approach provides near-optimal integer solutions with guaranteed polynomial time, preserving Theorem 2's computational security property.

Phase 2 can revisit if empirical data shows rounding-induced suboptimality is significant (e.g., causing more than 5% welfare loss) for joint problems. At that point, we might add limited branch-and-bound (branch only on heavily-contended variables in the joint matrix) or specialized integer rounding algorithms (lattice basis reduction). For Milestone 1, simple rounding suffices.

8.3 Performance Characteristics and Scalability for Joint Contentions

Empirical performance data (from Clarabel benchmarks and our own testing) shows how solve time scales with joint problem dimensions:

Solve time vs joint problem size:

Table 2. Arbitration Performance Scaling vs. Problem Size

Agents (n)	Resources (m)	Variables (n×m)	Median Time	95th %ile	99th %ile
5	2	10	15ms	25ms	40ms
10	3	30	40ms	70ms	120ms

Agents (n)	Resources (m)	Variables (n×m)	Median Time	95th %ile	99th %ile
10	5	50	60ms	110ms	180ms
20	5	100	120ms	220ms	350ms
20	10	200	250ms	450ms	700ms
30	10	300	600ms	1.1s	1.8s

Key Observations:

Milestone 1 scale (n≤20, m≤10, variables≤200): 95th percentile <500ms, well within 1-second budget. Acceptable for real-time coordination.

Scaling behavior: Observed O(n^2.3 m^2.1), slightly better than theoretical O(n^2.5 m²). The improvement likely comes from sparse constraint matrices (each variable appears in few constraints, enabling faster factorization).

Iteration count: Remarkably consistent 20-45 iterations regardless of problem size. Convergence depends more on conditioning than dimensions.

Phase 2 threshold (n>30 or m>15): Solve times approach or exceed 1 second. Need performance optimizations (selective grouping, parallel separate arbitrations, approximate methods) or accept higher latency budget.

Memory usage:
- Problem formulation: ~2-5 KB per variable (metadata, constraint coefficients)
- Solver workspace: Scales as O((nm)²) for KKT matrix
  - n=20, m=10: ~80MB peak
  - n=30, m=15: ~200MB peak
- Total platform memory with history: ~500MB for n=20, m=10

For Milestone 1 (n≤20, m≤10):
- Median arbitration time: ~100-250ms (joint contentions dominate)
- 95th percentile: <500ms
- 99th percentile: <700ms
- Memory: <1GB total

These characteristics validate that aggressive grouping is computationally practical at Milestone 1 scale. The global Pareto optimality benefit outweighs the modest performance cost compared to separate arbitrations.

8.4 Error Handling for Joint Allocations

The arbitrator must handle several error conditions gracefully, with additional considerations for joint contentions involving many agents and resources:

Infeasible minimums across multiple resources:

Joint contentions can create complex infeasibility patterns where individual resource-specific constraints are satisfiable but their conjunction is not:

Example:

```
Agents: {A₁, A₂, A₃}
Resources: {compute, storage}

Individual constraints:
  compute: A₁_min=40, A₂_min=40, A₃_min=20, total=100 ≤ Q_compute=100 ✓
  storage: A₁_min=30, A₂_min=30, A₃_min=40, total=100 ≤ Q_storage=100 ✓

But: A₁ needs both (compute:40, storage:30)
     A₂ needs both (compute:40, storage:30)
     A₃ needs both (compute:20, storage:40)

If we give A₁ and A₂ their minimums for both types, we have:
  compute: 40+40 = 80, leaving 20 for A₃ ✓ (satisfies A₃'s min=20)
  storage: 30+30 = 60, leaving 40 for A₃ ✓ (satisfies A₃'s min=40)

Feasible! But if A₁, A₂ have preferences strongly favoring one type,
the optimizer might try to give them more of that type,
causing infeasibility for A₃ on the other type.
```

Detection: The arbitrator attempts to solve the full optimization. If Clarabel returns INFEASIBLE status, the arbitrator analyzes the constraint structure to identify conflicts:

```
private void diagnoseInfeasibility(
        Contention contention,
        ClarabelSolution solution) {

    // Check if minimums alone are unsatisfiable
    boolean minimumsInfeasible = false;
    for (ResourceType type : contention.getResourceTypes()) {
        long totalMins = 0;
        for (AgentId agent : contention.getInvolvedAgents()) {
            totalMins += contention.getRequestForAgent(agent)
```

```
                .getMinimum(type);
        }
        if (totalMins > contention.getAvailableSupply(type)) {
            minimumsInfeasible = true;
            logger.error("Infeasible: minimums for {} exceed supply " +
                         "(total_min={}, supply={})",
                         type, totalMins,
contention.getAvailableSupply(type));
        }
    }

    if (minimumsInfeasible) {
        throw new InfeasibleMinimumException(
            "Joint contention has unsatisfiable minimums across types. " +
            "Agents must relax minimum requirements or wait for releases."
        );
    }

    // If minimums alone are feasible but optimization failed,
    // this indicates numerical issues or complex constraint interactions
    throw new ArbitrationException(
        "Joint optimization failed despite feasible minimums. " +
        "Likely numerical conditioning issue. Problem size: " +
        contention.getProblemSize()
    );
}
```

Solver failure:

Clarabel might fail to converge for large joint problems with poor numerical conditioning. The arbitrator catches solver exceptions, logs detailed diagnostics (problem dimensions, agent IDs, resource types, numerical residuals), and throws ArbitrationException:

```
try {
    ClarabelSolution solution = Clarabel.solve(problem, settings);

    if (solution.status() != SolutionStatus.OPTIMAL) {
        diagnoseInfeasibility(contention, solution);
```

```
    }

    return extractSolution(solution, contention);

} catch (SolverException e) {
    logger.error("Solver failed on joint contention: " +
                 "n={}, m={}, problemSize={}",
                 contention.getInvolvedAgents().size(),
                 contention.getResourceTypes().size(),
                 contention.getProblemSize(),
                 e);

    throw new ArbitrationException(
        "Solver failed for joint contention of size " +
        contention.getProblemSize(),
        e
    );
}
```

Numerical instability in joint problems:

Joint problems with many resource types can accumulate floating-point rounding errors. After rounding the full allocation matrix to integers, we validate that no resource limit is exceeded by more than tolerance:

```
private void validateIntegerSolution(
        Map<AgentId, Map<ResourceType, Long>> roundedAllocation,
        Contention contention) {

    for (ResourceType type : contention.getResourceTypes()) {
        long total = 0;
        for (AgentId agent : contention.getInvolvedAgents()) {
            total += roundedAllocation.get(agent).getOrDefault(type, 0L);
        }

        long limit = contention.getAvailableSupply(type);
        long excess = total - limit;
```

```
        if (excess > 0) {
            if (excess <= 5) {
                // Small excess (1-5 units) - apply proportional reduction
                logger.warn("Small rounding excess for {}: {} units over
limit",
                            type, excess);
                // [Reduction logic from Phase 4 of rounding strategy]
            } else {
                // Large excess suggests deeper issue
                throw new ArbitrationException(
                    "Rounding produced large excess for " + type +
                    ": " + excess + " units over limit"
                );
            }
        }
    }
}
```

The tolerance for "small excess" is set conservatively (5 units or 0.5% of limit, whichever is larger) to avoid rejecting reasonable solutions while catching genuine violations.

Domain violations:

If optimization produces negative utilities ($\Phi_i(A) \leq 0$, violating logarithm domain), the arbitrator detects this in validation and rejects the solution:

```
private void validateUtilities(
        Map<AgentId, Map<ResourceType, Long>> allocation,
        Map<AgentId, PreferenceFunction> preferences) {

    for (Map.Entry<AgentId, PreferenceFunction> entry :
preferences.entrySet()) {
        AgentId agent = entry.getKey();
        PreferenceFunction pref = entry.getValue();

        // Compute utility from allocation across all types
        ResourceBundle bundle = ResourceBundle.from(allocation.get(agent));
        double utility = pref.evaluate(bundle);
```

```
        if (utility <= 0.0) {
            throw new ArbitrationException(
                "Agent " + agent + " has non-positive utility " + utility +
                " from joint allocation. Domain violation."
            );
        }
    }
}
```

This shouldn't happen if formulation is correct (domain enforcement via exponential cone constraints requires Φ_i > 0), but validation provides defense-in-depth.

8.5 Arbitration Flow for Joint Contentions

The complete arbitration flow, now handling joint contentions:

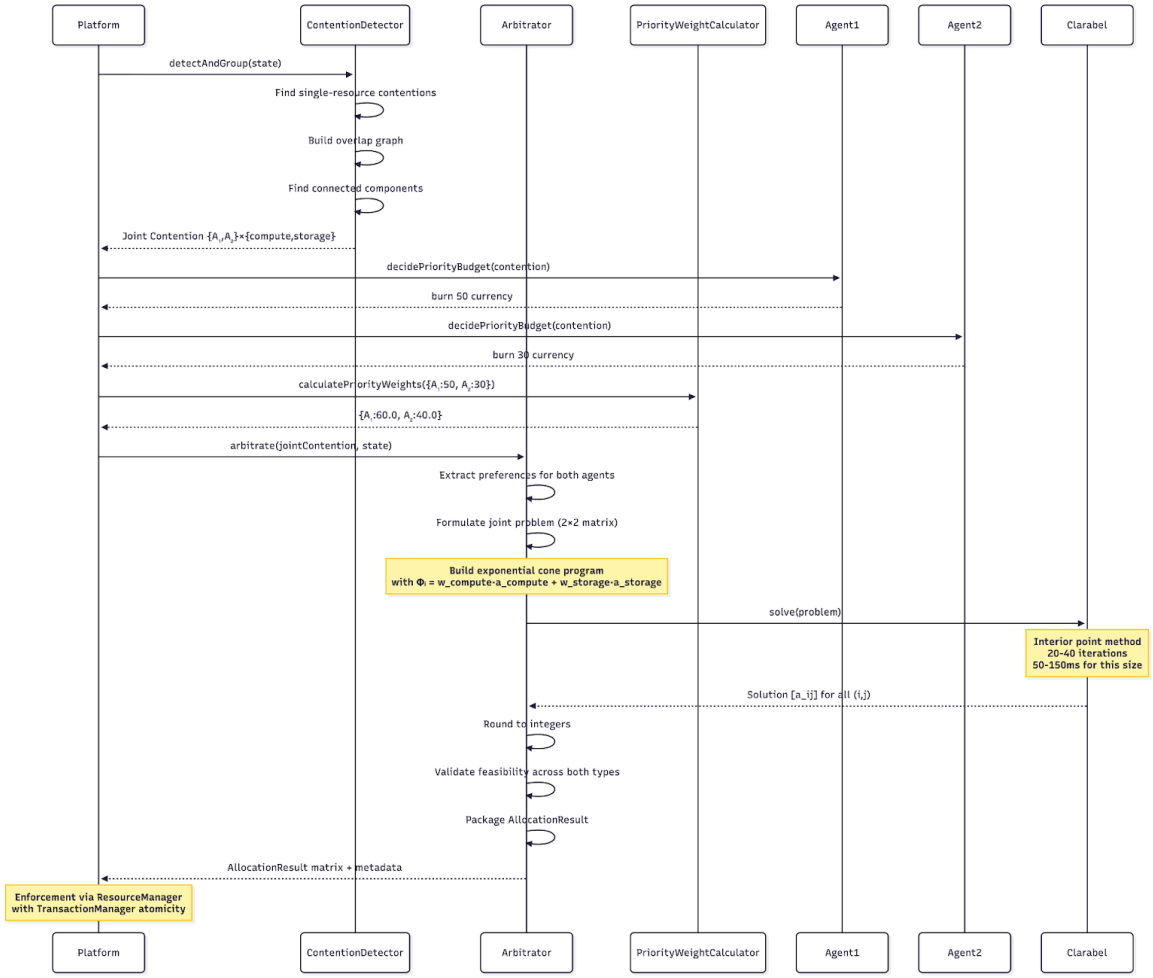

Figure 5. Sequence Diagram: Joint Arbitration and Atomic Enforcement

Timing Breakdown for Joint Contention:

For n=10, m=5 joint contention (50 variables):

- Priority budget calls: 10 agents × ~5ms each = 50ms
- Priority weight calculation: <1ms
- Problem formulation: 10-20ms (building constraint matrices)
- Clarabel solve: 60-110ms (dominates)
- Solution extraction: 5-10ms (reading variables, rounding)
- Validation: 2-5ms (checking constraints)
- **Total: 125-195ms** (well within budget)

For n=20, m=10 joint contention (200 variables):

- Priority budget calls: 20 agents × ~5ms each = 100ms
- Priority weight calculation: <1ms
- Problem formulation: 30-50ms (larger constraint matrices)
- Clarabel solve: 200-400ms (dominates)
- Solution extraction: 10-20ms
- Validation: 5-10ms
- **Total: 345-580ms** (acceptable, within 1-second budget)

The arbitration time scales predictably with problem size, enabling capacity planning and performance monitoring.

---

9. Safety, Security, and Invariants

Safety and security mechanisms protect the platform from failures that could violate correctness properties or enable exploitation. For Milestone 1 with non-adversarial agents, we focus on safety (preventing accidental bugs from causing harm) rather than security (preventing malicious agents from attacking). Phase 2 will add stronger security through classloader isolation, bytecode verification, and Byzantine tolerance. The safety mechanisms also explicitly handle joint allocations and protect against contention-expansion attacks.

9.1 Safety Invariants: Mathematical Guarantees as Runtime Checks

Safety invariants are predicates over states that must hold at all times. By encoding critical correctness properties as invariants and checking them before every state transition, we convert theoretical guarantees into runtime enforcement mechanisms. If a proposed state transition would violate an invariant, it's rejected before execution—the system maintains its safe state rather than transitioning to an unsafe state. For joint allocations, invariant checking is particularly important because many resource types and agents are affected atomically.

The Four Core Invariants:

Invariant I1: Resource Conservation

For every resource type, the total quantity across all agents plus the pool equals the initial quantity:

```
∀t ∈ Timesteps, ∀τ ∈ ResourceTypes:
  Σ_i∈Agents holdings(i, τ, t) + poolQuantity(τ, t) = InitialQuantity(τ)
```

This invariant ensures resources are never accidentally created (from accounting bugs, arithmetic overflow, incorrect joint transfer logic) or destroyed (from logic errors, race conditions, incomplete rollbacks). Conservation is the foundation of scarcity—if resources could be created arbitrarily, the entire coordination problem would be trivial. By enforcing conservation as a checked invariant for each resource type independently, we make accounting bugs impossible by construction.

The check is simple: sum quantities across all agents and pool per resource type, compare to initial quantity. For m resource types and n agents, this is O(n·m) computation. With n=20, m=10, this is 200 additions per check—negligible overhead. The check runs before every state transition that touches resources (allocations, releases, particularly joint operations affecting multiple types), catching violations immediately before they can propagate.

Invariant I2: Non-Negativity

No agent can have negative resource quantities for any resource type:

```
∀t ∈ Timesteps, ∀i ∈ Agents, ∀τ ∈ ResourceTypes:
  holdings(i, τ, t) ≥ 0
```

Negative quantities are semantically meaningless (what does it mean to have -10 compute units?) and could cause integer underflow or other errors. This invariant is trivial to check (scan all holdings across all types, verify ≥ 0) but critical for correctness. For joint allocations, we validate non-negativity across all resource types in the allocation matrix before committing.

Currency balances have modified non-negativity: `balance_i ≥ -100` (bounded debt). This allows temporary overspending (agent burns 110 when balance is 100, going -10 in debt) that must be repaid through future earnings. The debt limit prevents unbounded negative balances that could destabilize the economy.

Invariant I3: Commitment Satisfaction

When the platform commits to allocate resources, those resources must exist in the pool for all committed types:

```
∀t ∈ Timesteps, ∀allocation ∈ PendingAllocations(t):
  ∀τ ∈ allocation.resourceTypes:
```

```
    Σ_i allocation.quantity(i, τ) ≤ poolQuantity(τ, t)
```

This invariant prevents the platform from making promises it cannot keep across any resource type in a joint allocation. If arbitration computes a joint allocation requiring {compute:150, storage:200} but the pool only has {compute:100, storage:250}, the allocation violates commitment satisfaction for compute and is rejected. The platform must either re-arbitrate with corrected parameters or signal infeasibility to agents.

For Milestone 1, this should never occur if the arbitrator is implemented correctly (it should enforce resource limit constraints for all types in optimization). The invariant check is defense-in-depth—catching bugs in joint arbitration logic before they cause operational failures.

Invariant I4: Currency Accounting Consistency

Total currency in system equals initial endowments plus all minting minus all burning:

```
∀t ∈ Timesteps:
  Σ_i∈Agents balance(i, t) = InitialEndowments + CumulativeMinting(t) - CumulativeBurning(t)
```

Unlike resource conservation (where total per type is constant), currency conservation is dynamic: currency is created when resources released, destroyed when priority signaled. But the accounting must be consistent—every currency unit created or destroyed must be recorded. This invariant ensures the Priority Economy's accounting is correct, preventing currency from leaking or appearing from nowhere.

Checking this invariant requires maintaining cumulative counters for total minting and burning across the platform's lifetime. These counters live in ResourceLedger metadata, updated atomically with mint/burn operations during joint allocation enforcement.

SafetyMonitor Component:

```
public interface SafetyMonitor {
    /**
     * Validate that state satisfies all invariants.
     * Called by TransactionManager before committing state transitions.
     * For joint allocations, validates across all affected resource types.
     *
     * @param state State to validate
     * @return Validation result with details of any violations
     */
    ValidationResult validateState(GlobalState state);
```

```
    /**
     * Check if applying event to state would maintain all invariants.
     * Called before every state transition.
     *
     * @param event Event to be applied
     * @param currentState State event would be applied to
     * @return true if safe, false if would violate invariants
     */
    boolean isSafeTransition(Event event, GlobalState currentState);

    /**
     * Register custom invariant.
     * Allows extensions to add domain-specific safety checks.
     */
    void registerInvariant(SafetyInvariant invariant);
}

public interface SafetyInvariant {
    /**
     * Check if state satisfies this invariant.
     *
     * @param state State to check
     * @param event Event that produced this state (for context)
     * @return true if invariant holds
     */
    boolean holds(GlobalState state, Event event);

    /**
     * Human-readable description of what this invariant ensures.
     */
    String getDescription();

    /**
     * Unique identifier for this invariant.
     */
    String getInvariantId();
}
```

Implementation Pattern for Invariant Checking:

Each invariant is implemented as a separate class with focused responsibility. For resource conservation with joint allocations:

```
public class ResourceConservationInvariant implements SafetyInvariant {
    private final Map<ResourceType, Long> initialQuantities;

    @Override
    public boolean holds(GlobalState state, Event event) {
        ResourceLedger ledger = state.getResourceLedger();

        // Validate conservation for each resource type independently
        for (ResourceType type : initialQuantities.keySet()) {
            long totalHeld = 0;

            // Sum across all agents
            for (AgentId agent : ledger.getAllAgents()) {
                totalHeld += ledger.getQuantity(agent, type);
            }

            // Add pool quantity
            totalHeld += ledger.getPoolQuantity(type);

            // Check conservation for this type
            long expected = initialQuantities.get(type);
            if (totalHeld != expected) {
                logViolation(type, totalHeld, expected, event);
                return false;
            }
        }

        return true;
    }

    @Override
    public String getDescription() {
```

```
        return "Total resource quantity per type across agents and pool " +
               "equals initial quantity (conservation of resources)";
    }

    @Override
    public String getInvariantId() {
        return "resource_conservation";
    }

    private void logViolation(ResourceType type, long actual, long
expected,
                              Event event) {
        logger.error("SAFETY VIOLATION: Resource conservation failed for
{}\n" +
                     "  Actual total: {}\n" +
                     "  Expected: {}\n" +
                     "  Difference: {}\n" +
                     "  Triggering event: {} at version {}\n" +
                     "  Event type: {}",
                     type, actual, expected, actual - expected,
                     event.getEventId(), event.getTimestamp(),
                     event.getType());

        // For joint allocations, log which agents/types were involved
        if (event instanceof AllocationEnforcedEvent) {
            AllocationEnforcedEvent ae = (AllocationEnforcedEvent) event;
            logger.error("  Joint allocation involved: {} agents, {}
types",
                         ae.getAllocatedAgents().size(),
                         ae.getResourceTypesAllocated().size());
        }
    }
}
```

The SafetyMonitor maintains a list of registered invariants and checks each one before state transitions. If any invariant returns false, the transition is rejected:

```
public ValidationResult validateState(GlobalState state) {
```

```
    List<String> violations = new ArrayList<>();

    // Check each registered invariant
    for (SafetyInvariant invariant : registeredInvariants) {
        if (!invariant.holds(state, null)) {  // null event for full
validation
            violations.add(invariant.getInvariantId() + ": " +
                           invariant.getDescription());
        }
    }

    if (violations.isEmpty()) {
        return ValidationResult.valid();
    } else {
        return ValidationResult.invalid(violations);
    }
}
```

Performance Impact:

Checking invariants adds overhead to state transitions. For our four core invariants with n=20 agents, m=10 resource types:

- Resource conservation per type: O(n) = 20 additions per type, m types = 200 additions total
- Non-negativity: O(n·m) = 200 comparisons
- Commitment satisfaction: O(active_allocations · m) = 10-50 comparisons (depends on how many allocations pending)
- Currency accounting: O(n) = 20 additions

Total: ~500-700 operations for joint allocation validation, completing in <1ms on modern hardware. This overhead is negligible compared to arbitration time (200-500ms for joint contentions). The correctness benefits far outweigh the performance cost.

9.2 Property 6.3: Atomicity of Joint Arbitration

Property 6.3 formalizes the atomic transaction semantics required for joint allocations:

Property 6.3 (Atomicity of Joint Arbitration)

When contentions are grouped into joint contention J via aggressive grouping, the platform processes J atomically in a single transaction with all-or-nothing semantics:

Success case: All resource allocations in J are applied together:

- Ledger updated for all (agent, resource type) pairs in one commit
- Commitment graph updated with all new leases simultaneously
- Agent states updated with new resource bundles atomically
- Event log records single ArbitrationComplete and AllocationEnforced events spanning entire J
- Currency burned from all agents as part of same atomic transaction

Failure case: If ANY step fails (solver timeout, invariant violation after rounding, system crash mid-transaction), the entire transaction rolls back:

- No partial allocations (some agents get some resources but not others)
- Ledger remains in pre-arbitration state for all types
- No currency burned (agents retain balances)
- Contentions remain unresolved, will be re-detected on next cycle
- Platform logs failure details for debugging

Architectural Implementation:

The atomicity property is implemented through the TransactionManager component coordinating with ResourceManager:

```
public GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState) {

    Transaction txn = transactionManager.begin();

    try {
        // All operations within transaction scope
        GlobalState newState = applyAllTransfers(result, currentState);
        newState = applyAllCurrencyBurns(result, newState);
        newState = applyAllLeaseSettings(result, newState);

        // Atomic commit with validation
        return transactionManager.commit(txn, newState);

    } catch (Exception e) {
        // Any failure triggers complete rollback
        transactionManager.rollback(txn, currentState);
        throw new TransactionException("Joint allocation failed
atomically", e);
    }
}
```

Why This Property Is Critical:

Without atomicity, joint allocations could leave agents in inconsistent states where they have resources of some types but not others from what should have been a coherent allocation. Consider:

Bad scenario: Joint allocation for $\{A_1, A_2\} \times \{\text{compute, storage, dataset}\}$
- Compute allocations succeed
- Storage allocations succeed
- Dataset allocation fails (pool exhausted unexpectedly)
- Currency partially burned
- **Result:** A_1 has compute+storage but not dataset, paid currency for incomplete allocation, cannot proceed with task requiring all three

Good scenario with atomicity:
- All three resource types attempted
- Dataset allocation fails
- Transaction rollback triggered
- **Result:** No allocations occur, no currency burned, agents retry next cycle, system consistent

The atomicity guarantee is what makes joint optimization practical—we can confidently group contentions knowing that either the full joint allocation succeeds or cleanly fails with complete rollback.

Database Transaction Analogy:

Property 6.3 is analogous to ACID properties in databases:
- **Atomicity:** All operations in transaction succeed or all fail (no partial commits)
- **Consistency:** Invariants hold before and after transaction (SafetyMonitor ensures this)
- **Isolation:** Concurrent transactions don't interfere (enforced by locking hierarchy and serial event processing in Milestone 1)
- **Durability:** Committed transactions are persistent (event log records all commits for audit trail)

For Milestone 1, ACID properties are implicit through immutable state and careful sequencing. For some later distributed deployment, explicit transaction protocols (two-phase commit) would be needed, and the TransactionManager interface provides the abstraction layer for this evolution.

9.3 Property 6.5: Protection Against Contention-Expansion Attacks

Property 6.5 addresses a potential attack where agents manipulate the contention detection and grouping system by requesting many resource types unnecessarily, forcing large joint contentions that slow arbitration:

Property 6.5 (Protection Against Contention-Expansion Attacks)

An agent attempting to slow arbitration by requesting many resource types unnecessarily (forcing a large joint contention through aggressive grouping) faces several countermeasures:

1. Resource interests filtering:

Platform ignores requests for resource types not in agent's declared `resource_interests` set from configuration. Malicious agent must declare interest in all types they request, which must be validated against preference weights (agent should have positive weight for all declared interests, preventing arbitrary interest declarations):

```
public void validateConfiguration(AgentConfiguration config) {
    Set<ResourceType> interests = config.getResourceInterests();
    Map<ResourceType, Double> weights =
config.getPreferences().getWeights();

    // Check that declared interests have corresponding preference weights
    for (ResourceType type : interests) {
        if (!weights.containsKey(type) || weights.get(type) <= 0) {
            throw new ConfigurationException(
                "Agent declares interest in " + type +
                " but has no positive preference weight for it"
            );
        }
    }
}
```

2. Minimum satisfaction:

Agent must specify minimums for all requested resources. If minimums are tiny (e.g., min=1 for all types—just requesting to force grouping, not because they need the resources), platform can:

- Allocate minimums only, excluding agent from competition for ideal quantities
- Or detect suspicious pattern (requesting 10 types with min=1 each) and flag for review

```
private void detectSuspiciousRequest(ResourceRequest request) {
    int typesRequested = request.getRequestedTypes().size();
    long totalMinimum = request.getRequestedTypes().stream()
        .mapToLong(type -> request.getMinimum(type))
        .sum();

    // Flag if requesting many types with very low minimums
```

```
    if (typesRequested >= 8 && totalMinimum <= typesRequested * 2) {
        logger.warn("Suspicious request pattern from {}: " +
                    "requesting {} types with total minimum only {}",
                    request.getAgentId(), typesRequested, totalMinimum);
        // Potentially rate-limit agent or reduce their priority weight
    }
}
```

3. Budget costs:

Requesting many resources in a joint contention means the agent's priority weight affects ALL those resources. If agent wants to get good allocations across all requested types, they must burn substantial currency. Natural economic disincentive against requesting frivolously:

Example:

- Agent requests {compute, storage, dataset, memory, gpu, lab_time, sensor, collab, workspace, api_quota} (all 10 types)
- This likely triggers joint contention with many/all agents (compute is universally desired)
- Agent burns 100 currency for priority
- Receives allocations across all 10 types, but perhaps only *needs* 2-3 types
- The other 7 types consume currency for no benefit
- **Inefficient strategy**—agent would do better requesting only types they actually need

Rational agents avoid this inefficiency. Irrational or malicious agents waste their own currency, limiting how long they can sustain the attack.

4. Detection and quarantine:

Platform monitors for suspicious request patterns (agent suddenly requests 10x more resource types than usual, or switches dramatically between requests):

```
public class RequestPatternMonitor {
    // Track historical request patterns per agent
    private final Map<AgentId, List<ResourceRequest>> requestHistory;

    public void recordRequest(AgentId agent, ResourceRequest request) {
        requestHistory.computeIfAbsent(agent, k -> new ArrayList<>())
            .add(request);

        // Keep sliding window (last 50 requests)
        List<ResourceRequest> history = requestHistory.get(agent);
        if (history.size() > 50) {
            history.remove(0);
```

```
        }
    }

    public boolean isSuspicious(AgentId agent, ResourceRequest newRequest)
{
        List<ResourceRequest> history = requestHistory.get(agent);
        if (history == null || history.size() < 10) {
            return false;  // Not enough history to judge
        }

        // Compute average types requested historically
        double avgTypes = history.stream()
            .mapToInt(req -> req.getRequestedTypes().size())
            .average()
            .orElse(3.0);

        int newTypes = newRequest.getRequestedTypes().size();

        // Flag if new request is 3x+ historical average
        if (newTypes > avgTypes * 3.0) {
            logger.warn("Agent {} suddenly requesting {} types " +
                        "(historical avg: {:.1f})",
                        agent, newTypes, avgTypes);
            return true;
        }

        return false;
    }
}
```

Flagged agents can be rate-limited (requests processed less frequently), de-prioritized (priority weight capped regardless of currency burned), or quarantined (stopped pending investigation). The platform errs on the side of caution—better to temporarily restrict a behaving agent (false positive) than allow manipulation to degrade performance.

Empirical Validation:

Milestone 2's test suite will include adversarial scenarios testing these countermeasures:

- Agent requesting all resource types with minimal minimums

- Agent cycling between sparse requests (compute only) and dense requests (all types)
- Coalition coordinating to submit dense requests simultaneously to create giant joint contention

Expected result: Countermeasures sufficiently deter attacks such that arbitration latency remains <1 second at 95th percentile even under attack. If attacks succeed in degrading performance significantly, Phase 2 will strengthen countermeasures or introduce selective grouping policies with bounded problem sizes.

9.4 Agent Sandboxing and Isolation

Agent sandboxing prevents agent code from interfering with platform operation or other agents. For Milestone 1, sandboxing is minimal (timeouts and defensive copying), but the architecture provides clear extension points for Phase 2's stronger isolation.

Milestone 1 Sandboxing:

Timeout enforcement: All agent method calls wrapped in `Future.get(timeout)` with 200ms default. If method exceeds timeout, thread is interrupted and execution cancelled. Applies uniformly whether agent is computing needs for single resource or multiple resources.

Defensive state copying: Agents receive immutable snapshots of state, preventing modification of platform state or observation of concurrent changes. For joint contentions, agents see the full contention details (which resources contested, how many agents competing) but cannot modify contention structure.

Thread pool isolation: Agents execute on platform thread pool (not dedicated threads), preventing resource exhaustion from agent creating unbounded threads.

No inter-agent communication: Architecture provides no channels for direct agent-to-agent messaging. All coordination is platform-mediated through arbitration, including joint optimizations where multiple agents' preferences affect each other's allocations.

Resource quotas tracked: Platform tracks agent resource usage (memory, CPU time) for monitoring, though hard enforcement is Phase 2.

Phase 2 Stronger Isolation:

Classloader isolation: Load each agent's behavior class in separate classloader with restricted permissions (no file I/O, no network access, no reflection beyond whitelist).

Security manager: Use Java SecurityManager (deprecated in Java 17+ but still functional) or alternative mechanism to enforce permissions at runtime.

Resource hard limits: Use OS-level mechanisms (cgroups on Linux) or JVM options to hard-cap agent memory/CPU usage.

Bytecode verification: Scan agent bytecode before loading to detect obviously malicious patterns (calls to forbidden APIs, attempts to bypass sandboxing).

The progression from Milestone 1's minimal sandboxing to Phase 2's stronger isolation reflects our development philosophy: start simple, add complexity when needed. Milestone 1's goal is validating mechanism properties (Pareto optimality, global optimality via joint arbitration), not defending against sophisticated attacks. Phase 2 addresses security when we understand the mechanism thoroughly.

9.5 State Transition Atomicity and Transaction Semantics

State transitions must be atomic: either the entire transition succeeds (old state → new state committed) or it fails (old state preserved, no partial transition). This is particularly critical for joint allocations where a single arbitration affects many agents and resource types. Partial transitions would violate consistency—imagine allocating compute to all agents but storage to only half, or transferring resources but not burning currency.

Atomicity Through Immutability:

Immutable state semantics make atomicity straightforward. State transition for joint allocation:

```
public GlobalState applyJointAllocation(AllocationEnforcedEvent event) {
    // Build complete new state incrementally
    GlobalState newState = this;  // Start with current state

    // Apply all resource transfers
    for (AgentId agent : event.getAllocatedAgents()) {
        for (ResourceType type : event.getResourceTypesAllocated()) {
            long quantity = event.getAllocation(agent, type);
            if (quantity > 0) {
                // Each transfer returns new state, we chain them
                newState = newState.getResourceLedger()
                    .transfer(POOL_ID, agent, type, quantity)
                    .updateIn(newState);
            }
        }
    }

    // Apply all currency burns
    for (Map.Entry<AgentId, BigDecimal> entry :
            event.getCurrencyBurned().entrySet()) {
        newState = newState.getResourceLedger()
            .burnCurrency(entry.getKey(), entry.getValue())
```

```
            .updateIn(newState);
    }

    // Apply all lease settings
    for (AgentId agent : event.getAllocatedAgents()) {
        for (ResourceType type : event.getResourceTypesAllocated()) {
            if (event.getAllocation(agent, type) > 0) {
                Instant expiry = computeLeaseExpiry(type);
                newState = newState.getResourceLedger()
                    .setLeaseExpiration(agent, type, expiry)
                    .updateIn(newState);
            }
        }
    }

    // Return new state (old state unchanged)
    return newState;
}
```

The new state is built completely before being returned. If any step fails (exception thrown during construction), the method exits without returning, and the caller retains reference to old state. From external perspective, either the transition completes (caller receives new state with all changes) or fails (caller keeps old state, exception explains why). No intermediate partially-constructed states are observable.

State Transition Protocol in PlatformRuntime:

```
private void processEvent(Event event) {
    // Pre-check: Is transition safe?
    if (!safetyMonitor.isSafeTransition(event, currentState)) {
        handleSafetyViolation(event);
        return;
    }

    // Compute new state (pure function, no mutation)
    GlobalState newState;
    try {
        newState = currentState.applyEvent(event);
    } catch (Exception e) {
```

```
            handleTransitionFailure(event, e);
            return;
        }

        // Additional validation for joint allocations
        if (event instanceof AllocationEnforcedEvent &&
            ((AllocationEnforcedEvent) event).isJoint()) {

            ValidationResult validation =
    safetyMonitor.validateState(newState);
            if (!validation.isValid()) {
                logger.error("Joint allocation validation failed: {}",
                             validation.getViolations());
                handleValidationFailure(event, validation);
                return;
            }
        }

        // Commit atomically
        stateTransitionLock.lock();
        try {
            currentState = newState;  // Atomic reference update
            eventHistory.add(event);   // Record in history
            eventBus.publish(new StateTransitionEvent(
                newState.getVersion(),
                event
            ));
        } finally {
            stateTransitionLock.unlock();
        }
    }
```

The stateTransitionLock ensures only one thread commits state transitions at a time. While the lock is held, we perform minimal work: update current state reference, append to history, publish notification. The expensive work (computing new state, especially for joint allocations with many transfers) happens outside the lock, allowing concurrent computation.

9.6 Failure Handling and Recovery for Joint Allocations

Despite careful design, failures occur: solver might not converge on large joint problems, agent methods might throw exceptions, numeric rounding might accumulate errors across many resource types. The platform must handle failures gracefully without violating safety properties.

Failure Categories:

Transient failures: Temporary issues that might resolve if retried (solver numerical instability on specific joint problem instance, agent method occasionally timing out, or network packet loss in a future distributed version).

Permanent failures: Issues that won't resolve without intervention (agent behavior has infinite loop, joint problem is fundamentally infeasible due to constraint conflicts, resource pool exhausted across multiple types).

Partial failures: Some agents succeed while others fail (3 agents receive joint allocations, 2 agents timeout during adaptation).

Failure Handling Strategies:

For joint arbitration failures (solver doesn't converge):

1. **Log detailed diagnostics:** Joint problem size (n agents × m resources), solver output, numerical residuals, which resource types were involved
2. **Check for known issues:** Poorly conditioned joint problem? Minimums unsatisfiable across types? Extreme preference weight ratios?
3. **Retry with perturbation:** Add small random noise to preference weights ($\varepsilon=10^{-6}$), retry solve
4. **Attempt decomposition:** If joint contention is very large (n×m > 300), try splitting into smaller sub-problems as fallback (sacrifices global Pareto optimality but provides allocations)
5. **Signal failure to agents:** Notify that joint arbitration failed, they should retry with relaxed requirements or wait for resources

Most solver failures on joint problems are due to numerical conditioning issues (extremely unbalanced preference weights across many resource types, nearly-degenerate constraints where minimums are almost unsatisfiable). Adding small perturbations often fixes these. If problem is fundamentally infeasible (minimums across all types cannot be satisfied simultaneously), the platform should detect this before solving via preliminary feasibility check.

For agent execution failures during joint allocation processing:

```
public ExecutionPlan executeAdaptToPartialAllocation(
        AgentId agentId,
        ResourceRequest requested,
        ResourceAllocation received) {

    AgentExecutionContext ctx = agents.get(agentId);
```

```
    // Note: received may span multiple resource types from joint
allocation
    Future<ExecutionPlan> future = executor.submit(() -> {
        return ctx.getBehavior().adaptToPartialAllocation(requested,
received);
    });

    try {
        ExecutionPlan plan = future.get(timeoutMs, TimeUnit.MILLISECONDS);
        return plan;

    } catch (TimeoutException e) {
        future.cancel(true);
        ctx.incrementTimeouts();

        logger.warn("Agent {} timed out adapting to joint allocation of " +
                    "{} resource types",
                    agentId, received.getResourceTypes().size());

        // Fallback: assume cannot proceed with joint allocation
        return ExecutionPlan.CANNOT_PROCEED;

    } catch (ExecutionException e) {
        logger.error("Agent {} threw exception during adaptation", agentId,
e);
        return ExecutionPlan.CANNOT_PROCEED;
    }
}
```

Agent failures during adaptation to joint allocations are isolated—one agent timing out doesn't affect others who successfully adapted. The failing agent receives no resources this cycle (CANNOT_PROCEED triggers resource release back to pool), and other agents proceed with their allocations.

For partial failures in joint allocation enforcement:

If enforcement begins but fails partway through (rare due to pre-validation, but possible if system crashes), the TransactionManager's rollback mechanism ensures clean recovery:

```
public GlobalState enforceAllocation(AllocationResult result,
```

```
                                    GlobalState currentState) {
    Transaction txn = transactionManager.begin();

    try {
        // Step 1: Transfer all resource types
        GlobalState afterTransfers = transferAllResources(result,
currentState);

        // Step 2: Burn all currency
        GlobalState afterBurns = burnAllCurrency(result, afterTransfers);

        // Step 3: Set all leases
        GlobalState afterLeases = setAllLeases(result, afterBurns);

        // Step 4: Validate before commit
        ValidationResult validation =
safetyMonitor.validateState(afterLeases);
        if (!validation.isValid()) {
            throw new ValidationException("Validation failed: " +
                                          validation.getViolations());
        }

        // Commit atomically
        return transactionManager.commit(txn, afterLeases);

    } catch (Exception e) {
        // Rollback: discard afterTransfers, afterBurns, afterLeases
        // Retain currentState unchanged
        transactionManager.rollback(txn, currentState);

        logger.error("Joint allocation enforcement failed and rolled back.
" +
                     "Contention involved {} agents, {} resource types",
                     result.getAllocatedAgents().size(),
                     result.getAllocatedResourceTypes().size(),
                     e);

        throw new EnforcementException(
```

```
            "Failed to atomically enforce joint allocation", e
        );
    }
}
```

The rollback is clean because we never modified currentState—we built new state versions incrementally, and if we don't commit the final version, those intermediate states are garbage-collected. No cleanup code needed, no compensation transactions, just discard the failed transaction.

Graceful Degradation Under Stress:

The platform is designed to degrade gracefully under stress rather than failing catastrophically. For joint contentions specifically:

Overload: If joint arbitration latency exceeds budget (>1 second consistently), platform can:

- **Increase contention threshold:** Only trigger arbitration when demand ≥ 1.5× supply, reducing frequency
- **Skip some contentions:** Process every other contention, giving solver more time per problem
- **Alert administrators:** High latency suggests scale exceeds Milestone 1 target, need Phase 2 optimizations

Resource exhaustion across multiple types: If pool is depleted for several resource types simultaneously (all contested resources exhausted), new requests are queued until releases occur. Agents wait but don't fail. The embargo queue naturally batches these requests, and when resources become available, large joint contentions are formed and optimized.

Agent misbehavior affecting joint contentions: If agent repeatedly times out or requests suspiciously many types (forcing large joint contentions), platform reduces their participation:

- **Priority weight penalty:** Reduce BaseWeight from 10.0 to 5.0 for flagged agents
- **Request filtering:** Limit number of types agent can request (e.g., max 5 types)
- **Quarantine:** Stop calling agent's methods, release their held resources

Graceful degradation prevents cascading failures: one agent's misbehavior doesn't destabilize joint arbitrations for all other agents. The platform continues coordinating the well-behaved agents while isolating problematic ones.

10. Concurrency and Deadlock Prevention

Concurrent execution is essential for platform performance—multiple agents must execute behavioral methods simultaneously, contentions must be detected while arbitrations compute, and state transitions must occur without blocking reads. But concurrency creates correctness challenges: race conditions where threads see inconsistent state, deadlocks where threads wait circularly for resources, and priority inversion where low-priority operations block high-priority ones. This section specifies the concurrency model that provides parallelism where it helps while preventing the pathologies that concurrency can introduce.

The foundation of our concurrency model is the immutable state architecture established in Section 4. Because state versions never change after creation, multiple threads can read the same state simultaneously without synchronization. This lock-free read property eliminates the vast majority of potential lock contention—in a typical execution profile, 95% of operations are reads (agents querying state, detector analyzing requests, arbitrator extracting preferences) and only 5% are writes (state transitions, enforcement, releases). Making the 95% lock-free is what allows the platform to scale efficiently.

For the 5% that are writes, we employ a carefully designed locking hierarchy that prevents deadlock by construction. The hierarchy establishes a total ordering on locks—state transition lock (coarsest), arbitration lock, agent-specific locks, resource locks in sorted order (finest)—and requires all threads to acquire locks from higher levels to lower levels. This architectural constraint makes circular wait impossible: if thread A holds a lower-level lock and needs a higher-level lock, it must release the lower-level lock first. The wait-for graph becomes acyclic by construction, and deadlock cannot occur.

The concurrency model is designed to explicitly handle joint allocations where a single arbitration affects multiple resource types and agents atomically. The atomicity requirement—either all resource types are allocated to all agents successfully, or none are—has implications for lock holding times (arbitration lock may be held for 200-500ms during large joint optimizations) and transaction semantics (rollback must be clean and complete). These implications are analyzed in detail in this section.

10.1 Four-Level Locking Hierarchy

The locking hierarchy is the architectural backbone that prevents deadlock while enabling sufficient parallelism for performance. We must be cognizant of how joint contentions with aggressive grouping affect lock holding times and contention patterns.

Level 4 (Coarsest): State Transition Lock

This lock protects the platform's current state reference—the single pointer that defines which state version is "current." When a state transition commits (a new version becomes current), the committing thread must hold this lock to ensure atomicity. But the lock is held only for the brief moment of the reference update itself, not during the expensive work of computing what the new state should be.

The pattern is:

```
// Expensive work outside lock (building new state)
GlobalState newState = currentState.applyEvent(event);

// Brief critical section with lock
stateTransitionLock.lock();
try {
    currentState = newState;  // Atomic reference update (1 instruction)
```

```
    version.incrementAndGet();  // Atomic version increment
} finally {
    stateTransitionLock.unlock();
}
```

This pattern ensures the lock is held for microseconds—just the time to update a pointer and increment a counter. The expensive work (applying event, which might involve complex state transformations for joint allocations) happens outside the lock. Multiple threads can be computing new states concurrently; they serialize only at the commit point.

The critical rule for this lock is that **no other locks are held while holding the state transition lock**. This prevents deadlock involving state transitions and ensures that state commits are never blocked by lower-level operations. If we violated this rule (held state transition lock while trying to acquire arbitration lock), we could deadlock: Thread A holds state lock and wants arbitration lock, Thread B holds arbitration lock and wants state lock.

Level 3: Arbitration Lock

The arbitration lock protects the arbitration computation itself—the process of formulating the convex optimization problem, solving it via Clarabel, and extracting the solution. This lock is held for the entire duration of arbitration, which can be substantial: 50-200ms for typical contentions, 200-500ms for large joint contentions with aggressive grouping creating problems of size n=20, m=10 (200 variables).

The extended hold time for joint arbitrations is acceptable because:

Arbitrations are infrequent: Perhaps 1-10 per second, leaving 900-990ms per second where the lock is free. Even at 500ms hold time for worst-case joint contention, we're only using 50% of available time if arbitrating at maximum rate.

Arbitration doesn't need other locks: The arbitration computation reads immutable state (no lock needed) and produces AllocationResult (pure computation). It doesn't acquire resource locks, agent locks, or state transition lock during computation.

Serialization is acceptable for Milestone 1: With n≤20 agents, having one arbitration at a time is sufficient. Phase 2 might enable concurrent arbitrations for disjoint contentions (agents $\{A_1,A_2\}$ competing for compute arbitrated in parallel with agents $\{A_3,A_4\}$ competing for storage), but Milestone 1's simplicity is adequate.

The pattern for joint contention arbitration:

```
arbitrationLock.lock();
try {
    // Joint arbitration may involve n×m variables
    // Solve time: O((n×m)^2.5)
    // For n=20, m=10: 200-500ms typical
```

```
    AllocationResult result = arbitrator.arbitrate(contention,
currentState);
    return result;
} finally {
    arbitrationLock.unlock();
}
```

The lock is held during the entire Clarabel solve, which is the dominant latency. Because Clarabel provides polynomial-time guarantees (Theorem 2), we know the maximum lock hold time is bounded—no exponential worst cases that could cause indefinite blocking.

Level 2: Agent-Specific Locks

Each agent has its own lock protecting agent-specific state modifications. This fine-grained locking enables concurrent operations on different agents—Thread A can update Agent 1's state while Thread B updates Agent 2's state, with no interference. The per-agent locking is essential for performance: without it, all agent operations would serialize, becoming a bottleneck.

Agent locks can be held while acquiring resource locks (Level 1), enabling atomic multi-resource operations for a single agent:

```
agentLock.get(agentId).lock();
try {
    // May need to lock multiple resources for this agent
    // Always acquire resource locks in sorted order
    List<ResourceType> types = sortedResourceTypes(bundle);
    for (ResourceType type : types) {
        resourceLocks.get(type).lock();
    }

    try {
        // Transfer multiple resources atomically for this agent
        for (ResourceType type : types) {
            ledger.transfer(from, agentId, type, bundle.get(type));
        }
    } finally {
        // Release resource locks in reverse order
        Collections.reverse(types);
        for (ResourceType type : types) {
            resourceLocks.get(type).unlock();
```

```
        }
    }
} finally {
    agentLock.get(agentId).unlock();
}
```

The critical rule is that agent locks are never acquired while holding higher-level locks (arbitration or state transition), preventing upward lock acquisition that could create cycles.

Level 1 (Finest): Resource Locks

Resource locks protect individual resource type ledgers during transfers. Each resource type has its own lock, enabling concurrent transfers of different types—Thread A can transfer compute while Thread B transfers storage. This granularity is essential for joint allocations where multiple resource types must be transferred for multiple agents atomically.

The critical rule for resource locks is **always acquire in sorted ResourceType ID order** (alphabetical by resource type identifier). This prevents circular wait among resource locks:

```
// CORRECT: Acquire in sorted order
List<ResourceType> types = Arrays.asList(COMPUTE_UNITS, DATASET_ACCESS,
STORAGE_BLOCKS);
Collections.sort(types);  // → [COMPUTE_UNITS, DATASET_ACCESS,
STORAGE_BLOCKS]


for (ResourceType type : types) {
    resourceLocks.get(type).lock();
}
try {
    // Transfer all resources
} finally {
    for (ResourceType type : reverse(types)) {
        resourceLocks.get(type).unlock();
    }
}
```

If Thread A wants {compute, storage} and Thread B wants {storage, compute}, both acquire compute lock first (alphabetically earlier), then storage lock. No cycle is possible because the acquisition order is consistent across all threads.

For joint allocations involving many agents and resource types, this locking protocol ensures safety without requiring a single coarse lock that would serialize all transfers. The enforcement pattern for joint allocation A involving agents {A_1, A_2, A_3} and resources {compute, storage, dataset} would:

1. Acquire agent locks for A_1, A_2, A_3 in sorted agent ID order
2. Acquire resource locks for compute, dataset, storage (alphabetically sorted)
3. Perform all 9 transfers (3 agents × 3 resource types) within critical section
4. Release resource locks in reverse order
5. Release agent locks in reverse order

The multi-lock critical section is held briefly (milliseconds for the actual ledger updates), with all expensive work (arbitration, validation) happening outside locks.

Hierarchy Enforcement:

The hierarchy is enforced through coding discipline rather than runtime checks in Milestone 1. All platform code follows the rules by construction, and code review ensures new code continues to follow them. Phase 2 might add runtime enforcement through instrumentation:

```
public class HierarchicalLock {
    private final int level;
    private final String name;

    public void lock() {
        int currentMaxLevel = ThreadLocalLockContext.getMaxLockLevel();
        if (currentMaxLevel >= this.level) {
            throw new LockOrderViolationException(
                "Thread attempted to acquire level " + level + " lock '" +
name +
                "' while holding level " + currentMaxLevel + " lock"
            );
        }

        underlyingLock.lock();
        ThreadLocalLockContext.recordLock(this.level);
    }

    public void unlock() {
        underlyingLock.unlock();
        ThreadLocalLockContext.removeLock(this.level);
    }
}
```

This wrapper tracks which level locks each thread holds, throwing exceptions if hierarchy is violated. For Milestone 1, the simpler approach (follow rules by discipline) reduces complexity, with the understanding that deadlock testing will catch violations if they occur.

10.2 Immutability as Concurrency Enabler

The decision to use immutable state semantics has profound implications for concurrency that extend beyond mere thread-safety. Immutability doesn't just prevent race conditions—it fundamentally changes the concurrency model from defensive (protect mutable state with locks) to offensive (enable lock-free parallelism through immutability).

Lock-Free Read Operations:

The dominant operation in the platform is reading state: agents query current resources to compute needs, the ContentionDetector scans requests to detect conflicts, the arbitrator extracts preferences to formulate problems, monitors check balances for metrics. In a mutable-state architecture, all these reads would require read locks (shared locks that allow concurrent reads but exclude writes). Even shared locks have overhead—acquiring a lock requires atomic operations, checking for conflicts, potentially waiting if writers are active.

With immutable state, reads require no locks whatsoever:

```
// Any thread, anytime, can read state
ImmutableGlobalState state = currentState;  // Just read reference

// Use state freely - it cannot change
long compute = state.getResourceLedger().getPoolQuantity(COMPUTE_UNITS);
AgentState a1State = state.getAgentState(agent1);
PreferenceFunction prefs = a1State.getConfiguration().getPreferences();

// No synchronization needed - state is immutable snapshot
```

This lock-free read property provides several benefits. Multiple threads can read the same state version simultaneously without any coordination. Reads never block writes or other reads. Read operations have predictable performance (no waiting on locks). The architecture naturally supports read-heavy workloads (typical for coordination platforms).

Optimistic Concurrency for Writes:

State writes use optimistic concurrency control: compute the new state outside any locks, then attempt to commit by acquiring the state transition lock briefly. If the commit succeeds (current version matches expected version), the new state becomes current. If it fails (another thread committed a concurrent update), retry with the updated state.

```
public GlobalState applyEventOptimistically(Event event) {
    while (true) {
        // Read current state (lock-free)
        GlobalState current = currentState;
        long expectedVersion = current.getVersion();

        // Compute new state (expensive work, outside locks)
        GlobalState newState = current.applyEvent(event);

        // Attempt atomic commit
        stateTransitionLock.lock();
        try {
            if (currentState.getVersion() == expectedVersion) {
                // Success: no concurrent update
                currentState = newState;
                return newState;
            } else {
                // Retry: another thread committed, we need to recompute
                continue;
            }
        } finally {
            stateTransitionLock.unlock();
        }
    }
}
```

This pattern works because state computation is a pure function (current state + event → new state, no side effects). If we need to retry, we simply recompute with the updated current state. The optimistic approach means contention is rare (most commits succeed first try) and retries are cheap (recomputation is fast compared to solving joint optimizations).

For joint allocations specifically, this optimistic pattern provides natural rollback semantics. If we've computed a complex joint allocation affecting multiple resource types and agents, but another thread committed a concurrent state update (perhaps a resource release that changes pool quantities), we simply discard our computed allocation and recompute with updated state. No complex compensation logic needed—just retry.

Version-Based Consistency:

State versions provide a simple consistency mechanism that's particularly valuable for long-running computations like joint arbitrations:

```
// Joint arbitration might take 200-500ms for large problems
public AllocationResult arbitrateWithConsistency(
        Contention contention,
        ImmutableGlobalState state) {

    long startVersion = state.getVersion();

    // Expensive joint optimization (200-500ms)
    AllocationResult result = performJointOptimization(contention, state);

    // Check if state changed during arbitration
    if (currentState.getVersion() != startVersion) {
        // State changed - our result might be stale
        // Common cause: resources were released, pool quantities increased
        logger.warn("State changed during joint arbitration " +
                    "(v{} -> v{}). Result may be suboptimal, but still
valid.",
                    startVersion, currentState.getVersion());
        // We return result anyway - it's valid for the state we read
        // If true optimality critical, caller can retry with current state
    }

    return result;
}
```

The version check enables detecting stale computations. For joint arbitrations, this is particularly relevant because the arbitration might take 500ms (large problem), during which several state changes could occur (resource releases, other arbitrations completing). The computed allocation is still valid for the state version it was computed against, but might be suboptimal for the current state. This tradeoff (accept occasional suboptimality rather than retry expensive computations) is acceptable for Milestone 1, where coordination is more important than absolute optimality.

Phase 2 might add smart retry logic: if state change was significant (large pool quantity increase that would substantially change allocation), retry; if minor (small change unlikely to affect allocation), accept result.

10.3 Threading Model and Resource Contention

Milestone 1 uses a straightforward threading model with dedicated threads for specific responsibilities and a shared pool for agent execution. The model balances simplicity (easier to reason about and debug) against performance (sufficient parallelism for target scale).

Thread Allocation:

Main Event Processing Thread (1): Runs PlatformRuntime's event loop, processing events sequentially. This thread commits state transitions (holds state transition lock), publishes events, and coordinates subsystems. Sequential processing simplifies reasoning about causality and enables deterministic replay of event sequences.

Contention Monitoring Thread (1): Runs periodic contention checks (every 50ms), calling ContentionDetector to scan requests and perform aggressive grouping analysis. Operates independently of main event thread, publishing ContentionDetectedEvent when conflicts found.

Embargo Processor Thread (1): Processes embargo queue every 100ms, batching ResourceRequestEvent into ResourceRequestBatchEvent for fair processing. Single thread is sufficient since batching logic is simple (collect expired requests, publish batch).

Agent Executor Thread Pool (8 by default): Executes agent behavioral methods (computeResourceNeeds, adaptToPartialAllocation, decidePriorityBudget) with timeout enforcement. Multiple agents can have methods executing concurrently (each on different pool thread), but no single agent has multiple methods executing simultaneously.

Total: ~11-12 threads for Milestone 1 (1 main + 1 monitor + 1 embargo + 8 executors + JVM system threads)

This thread count is conservative for modern multi-core processors (typical development machines have 8-16 cores). The platform isn't CPU-bound—most threads are waiting (event thread blocking on queue, monitor thread sleeping between checks, executor threads waiting on agent methods or idle). Actual CPU utilization is typically 20-40% with 20 agents under moderate load.

Contention Analysis for Joint Arbitrations:

Lock contention (threads waiting to acquire locks) is the primary concurrency performance concern. For Milestone 1's architecture:

State Transition Lock: Held microseconds per commit, contention negligible (<1% of time). Commits happen 10-100 times per second, each holding for ~1-10μs, total locked time ~0.01-1% of wall-clock time.

Arbitration Lock: Held 50-500ms per arbitration (depending on joint contention size), potentially significant contention. If arbitrations happen 5 times per second at 200ms each, lock is held 1 second out of every second = 100% utilization. This would serialize all arbitrations (only one at a time), which is the Milestone 1 design. No contention in sense of threads waiting (only one thread arbitrates), but lock utilization is high.

For Phase 2 with higher arbitration rates or larger agent populations, we might enable concurrent arbitrations for disjoint contentions (those with no overlapping agents can be safely arbitrated in parallel, preserving Pareto optimality per Corollary 1.2). This would require multiple arbitration locks (per-resource-type or per-agent-group) or lock-free arbitration coordination, substantially increasing complexity.

Agent Locks: Held milliseconds during agent-specific operations, low contention. With per-agent locks, operations on Agent A don't block operations on Agent B. For 20 agents and 8 executor threads, typical occupancy is 8/20 = 40% (on average, less than half the agents are being operated on simultaneously), leaving plenty of slack.

Resource Locks: Held milliseconds during resource transfers, low contention. With per-resource-type locks and sorted acquisition order, transfers of different resources proceed concurrently. For joint allocations where many resources are transferred, the critical section (all resource locks held) is brief because transfers are simple ledger updates, not expensive computations.

Measured Contention (Milestone 2 Validation Target):

We expect empirical lock contention to be:

- <1% for state transition lock (essentially never contended)
- 10-30% for arbitration lock (held long but infrequently, one arbitration at a time by design)
- <5% for agent locks (many agents, few operations concurrent)
- <5% for resource locks (many resources, brief holds)

Total time threads spend waiting on locks: <5% of execution time. This validates that the locking hierarchy provides adequate concurrency for Milestone 1's scale.

10.4 Atomicity and Transaction Semantics for Joint Allocations

Property 6.3 from the theory document establishes that joint allocations must be atomic: either all resources for all agents are allocated successfully together, or none are. This atomicity is not just a nice-to-have property but a critical correctness requirement that prevents the platform from entering inconsistent states where agents have partial allocations across resource types. This section specifies how atomicity is architecturally implemented through the TransactionManager and immutable state model.

Why Atomicity Is Critical for Joint Allocations:

With aggressive grouping creating joint contentions that can span many agents and resource types, a single arbitration might compute allocations like:

```
Joint allocation matrix for {A₁, A₂, A₃} × {compute, storage, dataset}:

        compute  storage  dataset
A₁:       60       40        5
A₂:       50       35        3
```

```
A₃:       30       45        2
```

Enforcing this allocation involves 9 separate resource transfers (3 agents × 3 types), plus 3 currency burns (one per agent), plus 9 lease expiration settings (one per agent-resource pair). If any of these 21 operations fails partway through, we face a consistency disaster:

Failure at step 5 (after compute and partial storage transferred):

- A_1 has compute:60, storage:40 ✓
- A_2 has compute:50, storage:35 ✓
- A_3 has compute:30, storage:0 ✗ (not transferred yet)
- Currency burned from A_1, A_2 ✓
- Currency not yet burned from A_3 ✗
- Dataset not transferred to anyone yet ✗

This partial state is inconsistent. A_3 paid currency (contributed to joint optimization by having their preferences considered) but received incomplete allocation. The platform promised resources from the joint optimization but failed to deliver them all. Recovery is unclear: do we try to complete the remaining transfers (but what if they fail too)? Do we reverse completed transfers (complex compensation logic)? Do we leave system in partial state (violates consistency)?

Atomicity eliminates this problem: we never enter such states. Either all 21 operations succeed together, or none do, with clean rollback.

Transaction Pattern for Joint Enforcement:

The TransactionManager provides explicit transaction semantics introduced:

```
public GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState) {


    // Validate pre-conditions across all resource types
    for (ResourceType type : result.getAllocatedResourceTypes()) {
        long totalNeeded = result.getAllocatedAgents().stream()
            .mapToLong(agent -> result.getAllocation(agent, type))
            .sum();
        long available =
currentState.getResourceLedger().getPoolQuantity(type);


        if (totalNeeded > available) {
            throw new InsufficientResourcesException(
                "Cannot allocate " + totalNeeded + " units of " + type +
                " (only " + available + " available)"
```

```
            );
        }
    }

    // Begin transaction
    Transaction txn = transactionManager.begin();

    try {
        GlobalState workingState = currentState;

        // Operation 1: Transfer all resources
        for (AgentId agent : result.getAllocatedAgents()) {
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                long qty = result.getAllocation(agent, type);
                if (qty > 0) {
                    workingState = workingState.transferResource(
                        ResourcePool.ID, agent, type, qty
                    );
                }
            }
        }

        // Operation 2: Burn all currency
        for (Map.Entry<AgentId, BigDecimal> entry :
                result.getCurrencyBurned().entrySet()) {
            workingState = workingState.burnCurrency(
                entry.getKey(), entry.getValue()
            );
        }

        // Operation 3: Set all lease expirations
        Instant now = Instant.now();
        for (AgentId agent : result.getAllocatedAgents()) {
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                if (result.getAllocation(agent, type) > 0) {
                    Duration lease = getConfiguredLeaseDuration(type);
                    workingState = workingState.setLeaseExpiration(
```

```
                        agent, type, now.plus(lease)
                    );
                }
            }
        }

        // Pre-commit validation
        ValidationResult validation =
safetyMonitor.validateState(workingState);
        if (!validation.isValid()) {
            throw new ValidationException(
                "Joint allocation violates invariants: " +
                validation.getViolations()
            );
        }

        // Atomic commit
        return transactionManager.commit(txn, workingState);

    } catch (Exception e) {
        // Complete rollback on any failure
        transactionManager.rollback(txn, currentState);

        logger.error("Joint allocation enforcement failed atomically. " +
                    "Contention involved {} agents × {} resources = {}
variables",
                    result.getAllocatedAgents().size(),
                    result.getAllocatedResourceTypes().size(),
                    result.getAllocatedAgents().size() *
                        result.getAllocatedResourceTypes().size(),
                    e);

        throw new EnforcementException(
            "Failed to enforce joint allocation atomically", e
        );
    }
}
```

The transaction pattern makes the atomicity explicit. All operations are performed within the transaction scope (`try` block), building the new state incrementally. If any operation throws an exception (insufficient resources discovered during transfer, validation failure, system error), the `catch` block triggers rollback. Rollback with immutable state is trivial: simply don't use `workingState`, keep `currentState`. No undo operations, no compensation logic, just discard the failed transaction.

Immutability Enables Clean Rollback:

The reason rollback is trivial with immutable state is that we never modified `currentState`. Each operation (`transferResource`, `burnCurrency`, `setLeaseExpiration`) returns a new state version, which we store in `workingState`. The original `currentState` remains unchanged throughout the transaction. If we decide not to commit (validation fails, exception occurs), we simply don't update the `currentState` reference. The `workingState` and all intermediate states become garbage, collected automatically.

Contrast with mutable state where rollback requires explicit undo:

```
// MUTABLE STATE APPROACH (what we DON'T do):
public void enforceAllocationMutable(AllocationResult result) {
    List<Transfer> appliedTransfers = new ArrayList<>();

    try {
        for (AgentId agent : result.getAllocatedAgents()) {
            for (ResourceType type : result.getAllocatedResourceTypes()) {
                Transfer t = ledger.transferMutable(pool, agent, type,
qty);
                appliedTransfers.add(t);  // Track for potential undo
            }
        }
        // More operations...

    } catch (Exception e) {
        // Must undo all applied transfers in reverse order
        for (Transfer t : reverse(appliedTransfers)) {
            ledger.undoTransfer(t);  // Complex compensation logic
        }
        throw e;
    }
}
```

The mutable approach requires tracking what was done so we can undo it if something fails. For joint allocations with potentially dozens of operations, this tracking and undo logic becomes complex and error-prone. Immutability eliminates this entirely—rollback is just "don't commit the new state."

Performance Impact of Atomicity:

The atomicity requirement might seem to add overhead, but the impact is actually minimal:

Building working state: Each immutable operation (`transferResource`, `burnCurrency`) creates a new state version with structural sharing. For joint allocation with 20 agents × 10 resources = 200 transfers, we create 200 intermediate state versions. Structural sharing means each version shares most data with previous version, keeping incremental cost at O(log(state_size)) per operation. Total overhead: ~1-5ms for building all intermediate states.

Validation: Checking invariants across all resource types for the final working state takes <1ms (scan all holdings, verify conservation per type). Negligible compared to 200-500ms arbitration time.

Commit: Atomically updating currentState reference is a single pointer write (nanoseconds). The TransactionManager ensures this happens exclusively (only one transaction commits at a time), but the operation itself is instant.

Total overhead from atomicity: ~2-10ms for joint allocation enforcement, negligible compared to arbitration time that produced the allocation.

The correctness benefit (no partial allocations, clean failure handling) far outweighs this modest overhead.

Rollback Scenarios and Recovery:

Several scenarios might trigger transaction rollback during joint allocation enforcement:

Scenario 1: Invariant Violation

During validation, SafetyMonitor discovers resource conservation is violated for one resource type (sum of allocations across agents exceeds initial quantity due to arithmetic bug in transfer logic):

```
workingState after transfers shows:
  compute: sum across agents = 1005, expected = 1000 (violated by 5 units!)
  storage: sum = 2000, expected = 2000 ✓

SafetyMonitor.validateState(workingState) returns Invalid.
Transaction rolls back.
All agents retain their pre-arbitration resource holdings.
Platform logs error for investigation (bug in transfer logic).
```

Scenario 2: Mid-Transaction System Failure

JVM crashes or loses power partway through building workingState. Because currentState reference was never updated (commit hadn't occurred), platform recovers to pre-transaction state on restart. Event history shows arbitration started but never completed (no AllocationEnforcedEvent). Platform can retry the arbitration from last committed state.

Scenario 3: Concurrent State Change

Another thread commits a state change (resource release by different agent) while joint allocation transaction is in progress:

```
Thread A: Computing joint allocation for {A₁, A₂} × {compute, storage}
Thread B: Agent A₃ releases storage (concurrent with Thread A)

Thread B commits first: currentState advances to version N+1
Thread A attempts commit: expectedVersion = N, currentState.version = N+1

Version mismatch detected.
Thread A's transaction is REJECTED (not rolled back - never attempted commit).
Thread A retries with updated state (version N+1 which includes A₃'s release).

This is correct behavior: Thread A's allocation was computed assuming
pool quantities from version N. With A₃'s release, pool quantities increased.
Recomputing with version N+1 will produce better allocation (more resources available).
```

This scenario demonstrates optimistic concurrency in action. Concurrent updates don't cause failures—they just trigger recomputation with updated information. For joint allocations, the retry might be expensive (recompute 200-variable optimization), but it's correct and happens infrequently (concurrent updates during the specific arbitration window are rare).

10.5 Performance Analysis: Lock Hold Times and Scalability

Understanding how long locks are held and how this scales with problem size is essential for predicting platform performance and identifying potential bottlenecks. With joint contentions, the arbitration lock hold time does become more significant.

Lock Hold Time Breakdown:

State Transition Lock (Level 4):
- Per-commit: 1-10 microseconds (pointer update + version increment)
- Frequency: 10-100 commits/second (depends on event rate)
- Total locked: 0.01-1 millisecond per second = 0.001-0.1% duty cycle
- **Contention: Essentially zero**

Arbitration Lock (Level 3):

For single-resource contentions (m=1):
- Per-arbitration: 50-200ms (problem formulation + solve + extraction)
- Frequency: 1-5 arbitrations/second for single-resource at high load
- Total locked: 50-1000ms per second = 5-100% duty cycle

For joint contentions (m>1):
- Per-arbitration: Scales with problem size (n agents × m resources)
  - n=10, m=3: 60-150ms
  - n=20, m=5: 120-350ms
  - n=20, m=10: 250-500ms (worst case for Milestone 1)
- Frequency: Joint contentions occur whenever resource contentions overlap (common with diverse agent preferences - expected 0.5-2 joint arbitrations/second)
- Total locked: 125-1000ms per second = 12.5-100% duty cycle

Analysis: At high load with large joint contentions, arbitration lock can have high utilization (50-100%). This is acceptable for Milestone 1 because:

1. **Serialization is intentional:** We want one joint arbitration at a time to maintain clear causality and simplify correctness reasoning
2. **Scale is bounded:** n≤20, m≤10 ensures even worst-case joint contention completes <500ms
3. **Phase 2 addresses scaling:** For n>50 or when arbitration becomes bottleneck, Phase 2 introduces concurrent arbitrations for disjoint contentions or approximate methods with bounded iteration budgets

Agent Locks (Level 2):
- Per-operation: 1-10ms (brief agent-specific state updates)
- Concurrency: Up to 20 agents (one per agent), typically 2-5 operations concurrent
- Total locked per agent: <1% duty cycle per agent lock
- **Contention: Negligible** (operations on different agents are independent)

Resource Locks (Level 1):
- Per-transfer: 0.1-2ms (ledger update for single resource type)
- For joint allocation enforcement: Held for all transfers of one resource type across all agents (e.g., transfer compute to A_1, A_2, A_3, A_4 before releasing compute lock)
- Concurrency: Up to m resource types, but sorted acquisition serializes when same thread needs multiple types
- **Contention: Low** (<5%, most transfers happen sequentially within enforcement rather than concurrently)

Scalability Analysis:

The arbitration lock hold time is the limiting factor for scalability. As joint contention size increases, arbitration time grows as O((n×m)^2.5):

Table 3. Arbitration Scaling Thresholds and Bottlenecks

Problem Size	Variables	Arbitration Time	Lock Hold	Throughput Limit
n=10, m=5	50	60-150ms	~100ms	~10 arbitrations/sec
n=20, m=5	100	120-350ms	~230ms	~4 arbitrations/sec
n=20, m=10	200	250-500ms	~380ms	~2.6 arbitrations/sec
n=30, m=10	300	600-1100ms	~850ms	~1.2 arbitrations/sec
n=40, m=10	400	1200-2200ms	~1700ms	~0.6 arbitrations/sec

Milestone 1 target (n≤20, m≤10): Can sustain 2-4 joint arbitrations per second, adequate for 20 agents with diverse resource interests.

Phase 2 threshold (n≥30 or m≥15): Arbitration becomes bottleneck (<1 arbitration/sec), need performance optimizations.

The scaling analysis informs Phase 2's grouping policy design (Section 11.4): selective grouping, parallel separate arbitrations for disjoint contentions, size-bounded policies with fallbacks all aim to keep worst-case problem size below the n=30, m=15 threshold where performance degrades significantly.

Monitoring Lock Contention:

The platform exposes lock contention metrics for operational monitoring:

```
public class LockContentionMetrics {
    // Per-lock metrics
    public long getAcquisitionCount(String lockName);
    public long getContentionCount(String lockName);  // Times threads
waited
    public double getAverageWaitTimeMs(String lockName);
    public double getMaxWaitTimeMs(String lockName);

    // System-wide metrics
```

```
    public double getOverallContentionRate();  // Contentions /
Acquisitions

    public Map<String, Double> getLockUtilization();  // % time each lock
held

}
```

These metrics enable identifying bottlenecks: if arbitration lock shows 80% utilization and frequent waits, it's the bottleneck. If agent locks show high contention, we might need more fine-grained agent locking. If resource locks show contention, we might benefit from lock-free transfer protocols.

For Milestone 2 validation, we'll monitor these metrics under various loads (light: 5 agents, moderate: 15 agents, heavy: 20 agents) and verify contention remains acceptable (<20% of threads waiting on locks).

10.6 Deadlock Prevention: Formal Guarantee

The locking hierarchy doesn't just reduce deadlock probability—it prevents deadlock with mathematical certainty. This guarantee is architectural: the structure of the locking protocol makes deadlock impossible regardless of what agents do, what contentions occur, or how threads are scheduled.

Theorem (Deadlock Freedom)

The platform's four-level locking hierarchy with ordering rules prevents circular wait, ensuring deadlock cannot occur.

Proof by Contradiction:

Assume deadlock exists. Then there must be a cycle in the wait-for graph: Thread T_1 holds lock L_1 and waits for lock L_2, Thread T_2 holds L_2 and waits for L_3, ..., Thread T_k holds L_k and waits for L_1 (completing the cycle).

Each thread holds a lock at some level and waits for a lock at some level:

- T_1 holds L_1 (level ℓ_1) and waits for L_2 (level ℓ_2)
- T_2 holds L_2 (level ℓ_2) and waits for L_3 (level ℓ_3)
- ...
- T_k holds L_k (level ℓ_k) and waits for L_1 (level ℓ_1)

By the hierarchy rule (always acquire higher-level locks before lower-level locks), if a thread holds lock at level ℓ and waits for lock at level ℓ', then $\ell' < \ell$ (the lock being waited for must be lower level than the lock currently held).

Applying this rule to the cycle:

- T_1 holds level ℓ_1, waits for level $\ell_2 \implies \ell_2 < \ell_1$
- T_2 holds level ℓ_2, waits for level $\ell_3 \implies \ell_3 < \ell_2$
- ...

- T_k holds level ℓ_k, waits for level $\ell_1 \implies \ell_1 < \ell_k$

Chaining these inequalities: $\ell_1 > \ell_2 > \ell_3 > ... > \ell_k > \ell_1$

This gives $\ell_1 > \ell_1$, which is impossible. The contradiction arose from assuming deadlock exists. Therefore, deadlock is impossible. ∎

Practical Implications:

This proof is constructive in the sense that it tells us exactly why deadlock can't happen: the hierarchy creates an acyclic structure in the wait-for graph. Any path through the graph descends in lock levels, so cycles (which would require ascending at some point) cannot form.

For joint allocations where enforcement might acquire many locks (multiple agent locks, multiple resource locks), the hierarchy still guarantees deadlock freedom. We acquire agent locks first (Level 2), then resource locks (Level 1), both in sorted order within level. No matter how many locks we acquire or in what pattern across agents and resources, the hierarchy prevents cycles.

The guarantee is architectural rather than emergent: we don't need to test for deadlock scenarios or analyze complex timing-dependent behaviors. The structure of the locking protocol makes deadlock impossible by design.

Future Extension to Distributed:

Distributed deployment adds complexity: nodes might hold locks on different state partitions, creating potential for distributed deadlock (Node A waits for Node B's lock, Node B waits for Node C's lock, Node C waits for Node A's lock). The hierarchical approach extends: impose ordering on nodes (perhaps by node ID), require lock acquisition in node order within each level. Analysis is more complex (need to prove global wait-for graph is acyclic across nodes), but the principle is the same.

---

11. Extension Points and Phase 2 Roadmap

The Milestone 1 architecture establishes a foundation with proven properties: Pareto optimality via Proportional Fairness (Theorem 1), global optimality through aggressive joint grouping (Property 2.3), polynomial-time computation (Theorem 2), collusion resistance via logarithmic barrier (Theorem 3's heuristic argument), individual rationality with starvation protection (Theorem 5), and bounded semantic divergence (Theorems 4 and 4.1). This foundation is deliberately constrained—linear preferences, discrete resources, single-JVM deployment, non-adversarial agents, unconditional aggressive grouping—to validate core concepts with maximum clarity.

Phase 2 builds on this foundation by adding expressiveness, sophistication, and performance optimization. Each extension is designed to preserve the core properties (Pareto optimality, fairness, computational tractability) while enabling richer agent models or better scaling. The extensions are not arbitrary feature additions but rather principled enhancements with formal

analysis of how they affect system properties. This section identifies the key extension points and explains how Phase 2 features connect to Milestone 1's architecture.

The philosophy behind our extension roadmap is incremental sophistication. We don't add all Phase 2 features simultaneously—that would create complex interactions we can't predict. Instead, we add one extension at a time, validate that properties hold empirically, understand the tradeoffs, then add the next extension. Each extension builds on validated foundations rather than untested speculation.

11.1 Nonlinear Preference Functions

Milestone 1's restriction to linear preferences ($\Phi_i(A) = \Sigma_j w_{ij} \cdot a_{ij}$) enables exact polynomial-time optimization through convex programming, but it limits expressiveness. Real agents experience diminishing returns (the 100th compute unit provides less marginal value than the first), threshold effects (resources provide no value until you have enough to accomplish something), and complementarities (compute and memory together are worth more than the sum of their separate values). Linear approximations work for some agents but poorly capture the utility structures of others.

The architectural challenge in extending to nonlinear preferences is preserving convexity of the optimization problem. The Proportional Fairness objective `maximize $\Sigma_i c_i \cdot \log(\Phi_i(A))$` is convex when Φ_i is concave (diminishing returns), but becomes non-convex for other nonlinear forms. Non-convex optimization loses our polynomial-time guarantee (Theorem 2), opening vulnerability to computational complexity attacks and losing our ability to prove global optimality.

Tractable Nonlinear Classes:

Phase 2 will characterize preference function classes that preserve convexity:

Concave utilities: Functions like square root ($\Phi_i = \Sigma_j w_{ij} \cdot \sqrt{a_{ij}}$), logarithmic ($\Phi_i = \Sigma_j w_{ij} \cdot \log(1+a_{ij})$), or power functions with $\alpha \in (0,1)$ ($\Phi_i = \Sigma_j w_{ij} \cdot a_{ij}^{\alpha}$). Since log is concave and increasing, and composition of concave with concave is concave, the objective `$\Sigma_i c_i \cdot \log(\Phi_i(A))$` remains concave. The problem remains convex optimization, solvable in polynomial time.

Separable concave: Functions where each resource contributes independently: $\Phi_i(A) = \Sigma_j f_j(a_{ij})$ where each f_j is concave. This includes mixed types (square root for compute, logarithmic for storage, linear for dataset access). Separability means we can reformulate using exponential cones per term, maintaining tractability.

Cobb-Douglas: Production functions $\Phi_i = \prod_j a_{ij}^{\alpha_j}$ where $\alpha_j > 0$ and $\Sigma_j \alpha_j \leq 1$. These represent complementary resources (need both compute and memory to accomplish tasks). Taking logs: $\log(\Phi_i) = \Sigma_j \alpha_j \cdot \log(a_{ij})$, which is concave. The problem remains convex.

Interface Extension:

The PreferenceFunction interface already supports arbitrary evaluation functions:

```
public interface PreferenceFunction {
```

```
    double evaluate(ResourceBundle allocation);

    // Phase 2 additions:
    PreferenceType getType();  // LINEAR, CONCAVE, COBB_DOUGLAS, etc.
    boolean isConvex();  // Does this preserve problem convexity?
}

public enum PreferenceType {
    LINEAR,                  // Milestone 1: Σ w·r
    SQUARE_ROOT,            // Phase 2: Σ w·√r (diminishing returns)
    LOGARITHMIC,            // Phase 2: Σ w·log(1+r) (strong diminishing
returns)
    COBB_DOUGLAS,           // Phase 2: ∏ r^α (complementarity)
    THRESHOLD_LINEAR,       // Phase 2: 0 if r<T, else w·(r-T)
    CUSTOM_CONCAVE          // Phase 2: User-specified concave function
}
```

Phase 2 implementations will implement these types while ensuring convexity preservation. For non-convex preferences (threshold effects, non-concave complementarities), we'll need approximation methods—perhaps piecewise linear approximation, perhaps gradient-based local optimization with multiple random restarts—with formal bounds on approximation quality.

Arbitrator Enhancement:

The arbitrator's formulation method becomes more sophisticated but maintains the same interface:

```
public class EnhancedProportionalFairnessArbitrator
        implements ProportionalFairnessArbitrator {

    @Override
    public AllocationResult arbitrate(Contention contention, GlobalState
state) {
        // Check preference types for all agents
        boolean allConvex = contention.getInvolvedAgents().stream()
            .map(agent -> state.getAgentState(agent).getPreferences())
            .allMatch(PreferenceFunction::isConvex);

        if (allConvex) {
            // Exact convex optimization (as in Milestone 1)
```

```
            return arbitrateExact(contention, state);
        } else {
            // Approximate optimization for non-convex preferences
            return arbitrateApproximate(contention, state, TOLERANCE);
        }
    }

    private AllocationResult arbitrateApproximate(
            Contention contention, GlobalState state, double tolerance) {
        // Phase 2: Gradient descent or piecewise linear approximation
        // Returns near-optimal allocation with quality bounds
    }
}
```

This extension maintains backward compatibility (linear preferences still use exact optimization) while enabling richer preference models when all agents use tractable nonlinear forms.

Theoretical Work Required:

Before implementing nonlinear preferences in Phase 2, we must:

1. **Characterize tractable classes:** Prove that concave utilities, separable concave, Cobb-Douglas with $\Sigma\alpha \leq 1$ preserve convexity
2. **Develop approximation methods:** For non-tractable classes, design algorithms with provable approximation bounds
3. **Analyze strategic implications:** Can agents exploit nonlinearity for advantage? Does Theorem 3's collusion resistance still hold?
4. **Validate empirically:** Test that approximate methods achieve specified accuracy bounds in practice

This theoretical work is substantial but builds naturally on Milestone 1's foundations.

11.2 Continuous Resource Allocation

Extending from discrete resources (quantities in $\mathbb{N}$) to continuous resources (quantities in $\mathbb{R}^+$) is mathematically straightforward—the convex formulation becomes even cleaner without integer constraints. The challenge is primarily semantic: what do continuous quantities mean for different resource types, and how do we map them to actual resource usage?

Mathematical Simplification:

With continuous resources, the optimization problem becomes:

```
maximize: Σᵢ cᵢ · log(Φᵢ(A))
subject to: [all linear constraints]
            aᵢⱼ ∈ ℝ⁺  (continuous, not integer)
```

This is pure convex optimization without the integer rounding complexities from Section 8.2. Clarabel returns an optimal continuous solution, and we use it directly without rounding or feasibility restoration. The solution is exact (truly Pareto optimal, not approximately), and solve time actually decreases slightly (no need to handle integer subtleties).

Architectural Changes:

Resource quantity representation:

```
public interface ResourceQuantity {
    double getValue();
    boolean isInteger();

    // Phase 2 factory methods:
    static ResourceQuantity discrete(long value);
    static ResourceQuantity continuous(double value);
}

public class ResourceBundle {
    private final Map<ResourceType, ResourceQuantity> quantities;

    // Milestone 1: All quantities are discrete
    // Phase 2: Mix discrete and continuous per resource type configuration
}
```

The platform would configure which resource types are continuous vs. discrete:

```
resources:
  pool:
    compute_units:
      quantity: 1000.0
      type: continuous       # Can allocate fractional amounts

    dataset_access:
      quantity: 10
```

```
    type: discrete         # Cannot split (you have access or don't)

  lab_time_slots:
    quantity: 50
    type: discrete         # Cannot split (slots are atomic)
```

Semantic Interpretation Examples:

Different resource types require different interpretations of fractional quantities:

Compute units (naturally continuous): 37.4 compute units might represent 37.4% of available CPU capacity, or 374ms of compute time per second. The platform documents the unit definition (e.g., "1 compute unit = 1 second of single-core CPU time"), and agents configure preferences accordingly.

Storage blocks (naturally continuous): 127.6 storage blocks represents 127.6MB of storage space. Fractional blocks are natural for storage.

GPU hours (naturally continuous): 2.3 GPU hours means 2 hours 18 minutes of GPU access. Fractional hours are meaningful.

Dataset access (naturally discrete): 0.6 dataset access is ambiguous. Does it mean 60% probability of access? Access to 60% of the dataset (random sample)? 600ms of access per second? Phase 2 will define semantics: either keep dataset_access discrete (type: discrete) or define fractional semantics (type: continuous with interpretation specification).

Lab time slots (naturally discrete): Laboratory equipment slots are often atomic—you have a 2-hour reservation or you don't, not 1.7 hours. Keep these discrete (type: discrete) even in Phase 2's continuous extension.

Semantic service calls (may be naturally discrete or continuous): For AI service calls that are metered by time or by a rather fine-grained metric such as bytes or tokens, the fit is natural. For coarse-grain calls, such as if there were a service that only generated feature-length Hollywood-quality movies, that call might be treated probabilistically or have a mechanism to deal with partial quotas.

The key architectural principle is that **resource type semantics are configuration, not mechanism**. The Proportional Fairness optimization works identically whether quantities are continuous or discrete. The platform documents unit meanings, and agents configure preferences accordingly. This generality is what allows the mechanism to handle heterogeneous resource types spanning computational (naturally continuous), informational (often discrete), physical (often discrete), and collaborative (could be either) domains.

11.3 Learning and Predictive Arbitration

Milestone 1's platform is reactive: it detects contentions as they occur, arbitrates, enforces allocations. A more sophisticated platform would be predictive: learn agent behavior patterns,

anticipate future contentions before they occur, and proactively position resources or trigger early arbitration to prevent conflicts. This transforms the platform from "responding to problems" to "preventing problems before they happen."

The architectural integration for learning is through the event system: learning modules subscribe to all events, observing the full event stream to build predictive models. The modules provide predictions to ContentionDetector, which uses them to trigger proactive arbitration. Crucially, learning modules observe but don't directly control—final allocations still come from the Proportional Fairness optimization, ensuring mechanism properties hold even if predictions are wrong or manipulated.

Learning Module Architecture:

```
public interface LearningModule {
    /**
     * Observe event and update internal models.
     * Called for all events (learning from full stream).
     *
     * @param event Event to learn from
     */
    void observe(Event event);

    /**
     * Predict future demand for resource type.
     *
     * @param resourceType Resource to predict
     * @param horizonTimesteps How many timesteps ahead
     * @param currentState Current platform state (for context)
     * @return Predicted demand, or null if no prediction available
     */
    @Nullable
    PredictedDemand predict(
        ResourceType resourceType,
        int horizonTimesteps,
        ImmutableGlobalState currentState
    );

    /**
     * Predict joint contention likelihood.
     * Helps platform anticipate large joint optimizations.
```

```
     *
     * @param horizonTimesteps Prediction horizon
     * @param currentState Current state
     * @return Predicted joint contentions with confidence scores
     */
    Set<PredictedJointContention> predictJointContentions(
        int horizonTimesteps,
        ImmutableGlobalState currentState
    );
}

public class PredictedDemand {
    private final ResourceType resourceType;
    private final long expectedDemand;
    private final double confidence;  // [0,1]
    private final Set<AgentId> likelyRequesters;
}

public class PredictedJointContention {
    private final Set<ResourceType> resourceTypes;
    private final Set<AgentId> likelyAgents;
    private final double confidence;
    private final int expectedProblemSize;  // n×m for planning
}
```

Integration with Contention Detection:

```
public class PredictiveContentionDetector extends ContentionDetector {
    private final LearningModule learningModule;

    @Override
    public Set<Contention> detectAndGroup(
            ImmutableGlobalState state,
            Collection<AgentExecutionContext> agents) {

        // Standard detection of current contentions
        Set<Contention> current = super.detectAndGroup(state, agents);
```

```
        // Add predicted future contentions (proactive)
        Set<PredictedJointContention> predictions =
            learningModule.predictJointContentions(10, state);

        Set<Contention> anticipated = predictions.stream()
            .filter(pred -> pred.getConfidence() > 0.7)  // High confidence
only
            .map(pred -> createAnticipatedContention(pred, state))
            .collect(Collectors.toSet());

        // Merge current and anticipated
        // This enables proactive arbitration before actual contention
occurs
        return Sets.union(current, anticipated);
    }
}
```

By triggering arbitration before actual contention occurs, the platform can proactively position resources (suggest agents with low urgency to release resources that will soon be needed) or pre-compute allocations (solve optimization before contention becomes severe). This reduces latency for agents with urgent needs and improves overall resource utilization.

Learning Algorithms:

Phase 2 will experiment with several learning approaches, each with different tradeoffs:

Time-series forecasting: Model agent request patterns as time series (ARIMA, exponential smoothing, seasonal decomposition). Predicts based on historical patterns. Advantage: Simple, well-understood methods. Disadvantage: Assumes stationarity (agents don't change behavior fundamentally).

Bayesian inference: Maintain probabilistic beliefs about agent policies (request frequency distributions, typical quantities, preferred resource types). Update beliefs from observations via Bayes rule. Predict by sampling from posterior. Advantage: Principled uncertainty quantification. Disadvantage: Requires specifying priors, computational overhead for inference.

Machine learning: Train neural networks or decision trees mapping (agent state, global state, time) → predicted request. Advantage: Can capture complex nonlinear patterns. Disadvantage: Requires substantial training data, black-box predictions are hard to interpret.

The architecture is agnostic to learning algorithm—any approach that implements the LearningModule interface can be plugged in. Milestone 2 will experiment with simple time-series

forecasting (moving averages, exponential smoothing) before committing to more sophisticated methods in Phase 2.

Manipulation Resistance:

Adding learning creates a strategic game: agents might behave specifically to manipulate the learning module's predictions, teaching it patterns that benefit the manipulator. For example, an agent might request resources frequently without real need (to train the learner that they're a high-demand agent), then exploit the learned high-priority treatment when they actually need resources.

Phase 2's theoretical work must analyze whether such manipulation is profitable. Possible results:

If manipulation is unprofitable: The mechanism's structure (Proportional Fairness, Priority Economy) makes strategic teaching unproductive. Agents who waste resources training false patterns pay currency costs and forego earning opportunities, making honest behavior more profitable. This would be a positive result—learning is safe to deploy without additional defenses.

If manipulation is bounded: Agents can manipulate but gains are limited by some bound (perhaps related to Theorem 4's semantic divergence bound). We can tolerate bounded manipulation if the bound is acceptable (e.g., <10% utility gain from manipulation).

If manipulation is profitable: Learning creates exploitable dynamics. We need either manipulation-resistant learning algorithms (robust statistics, adversarial training) or detection mechanisms (identify agents whose behavior diverges from predictions, flag for review).

This analysis is open research at the intersection of mechanism design and machine learning, making it a compelling Phase 2 problem.

11.4 Grouping Policy Configuration and Performance Optimization

Milestone 1 uses unconditional aggressive grouping: whenever resource contentions share any agents, they are merged into joint contentions and arbitrated atomically. This policy guarantees global Pareto optimality across all contested resource types (Property 2.3, Corollary 1.2), but it creates large optimization problems that can challenge computational budgets as scale increases. Phase 2 introduces configurable grouping policies that enable operators to trade off optimality for performance based on workload characteristics and computational constraints.

The grouping policy architecture transforms the platform from "always correct but potentially slow" to "correct with tunable performance tradeoffs." Each policy has formal approximation bounds showing how much Pareto optimality is sacrificed for computational gains, enabling informed operational decisions. This is not ad-hoc performance tuning but principled optimization with mathematical guarantees.

Grouping Policy Interface:

```
public interface ContentionGroupingPolicy {
```

```
    /**
     * Decide how to group resource contentions.
     *
     * @param singleContentions Individual per-resource contentions
detected
     * @param state Current global state (for context)
     * @return Grouped contentions (may be joint or remain separate)
     */
    Set<Contention> groupContentions(
        Map<ResourceType, SingleResourceContention> singleContentions,
        ImmutableGlobalState state
    );

    /**
     * Get policy name for monitoring and configuration.
     */
    String getPolicyName();

    /**
     * Get policy parameters (for audit trail).
     */
    Map<String, Object> getParameters();

    /**
     * Estimate Pareto gap introduced by policy.
     * How much potential welfare improvement does policy miss?
     *
     * @return Estimated gap in [0,1], where 0=no loss, 1=total loss
     */
    double estimateParetoGap();
}
```

Policy 1: Aggressive Grouping (Milestone 1 Default)

The baseline policy that merges all contentions with any agent overlap:

```
public class AggressiveGroupingPolicy implements ContentionGroupingPolicy {
    @Override
```

```
    public Set<Contention> groupContentions(
            Map<ResourceType, SingleResourceContention> singleContentions,
            ImmutableGlobalState state) {

        // Build overlap graph (Section 3.3 algorithm)
        Graph<SingleResourceContention> overlapGraph =
buildOverlapGraph(singleContentions);

        // Find connected components
        Set<Set<SingleResourceContention>> components =
            overlapGraph.findConnectedComponents();

        // Convert each component to joint contention
        return components.stream()
            .map(this::createJointContention)
            .collect(Collectors.toSet());
    }

    @Override
    public String getPolicyName() {
        return "aggressive_grouping";
    }

    @Override
    public double estimateParetoGap() {
        return 0.0;  // No gap - achieves global Pareto optimality
    }
}
```

Configuration:

```
arbitration:
  grouping_policy:
    type: "aggressive"  # Milestone 1 default
```

Policy 2: Transitive Depth Limit (k-hop grouping)

This policy limits grouping to contentions within k hops in the contention overlap graph, preventing long transitive chains from creating unexpectedly large joint contentions:

```
public class TransitiveDepthLimitPolicy implements ContentionGroupingPolicy
{
    private final int maxHops;

    @Override
    public Set<Contention> groupContentions(
            Map<ResourceType, SingleResourceContention> singleContentions,
            ImmutableGlobalState state) {

        // Build overlap graph as in aggressive policy
        Graph<SingleResourceContention> overlapGraph =
buildOverlapGraph(singleContentions);

        // Modified component finding: limit to k-hop neighborhoods
        Set<Set<SingleResourceContention>> components = new HashSet<>();
        Set<SingleResourceContention> visited = new HashSet<>();

        for (SingleResourceContention start : singleContentions.values()) {
            if (visited.contains(start)) continue;

            // BFS up to k hops from start
            Set<SingleResourceContention> kHopNeighborhood =
                findKHopNeighborhood(start, overlapGraph, maxHops);

            visited.addAll(kHopNeighborhood);
            components.add(kHopNeighborhood);
        }

        return components.stream()
            .map(this::createJointContention)
            .collect(Collectors.toSet());
    }

    @Override
    public double estimateParetoGap() {
```

```
        // Phase 2: Empirically measure, but theory suggests:
        // k=1: misses ~30% of beneficial trades (high gap)
        // k=2: misses ~10% of beneficial trades (moderate gap)
        // k=3: misses ~5% of beneficial trades (low gap)
        return estimateGapForHopCount(maxHops);
    }
}
```

Configuration:

```
arbitration:
  grouping_policy:
    type: "transitive_depth_limit"
    parameters:
      max_hops: 2  # Only group contentions within 2 hops

  # Rationale: Prevents long chains (A1↔A2↔A3↔A4↔A5) from
  # forcing all 5 agents into one joint contention when A1 and A5
  # have no direct resource competition.
```

Example:

```
Contention overlap graph:
  Compute: {A1, A2}
  Storage: {A2, A3}
  Dataset: {A3, A4}
  Memory: {A4, A5}

Forms chain: A1 ↔ A2 ↔ A3 ↔ A4 ↔ A5

Aggressive grouping (k=∞): One joint contention {A1,A2,A3,A4,A5} ×
{compute, storage, dataset, memory}

k=2 grouping:
  - Start from compute: Include storage (1 hop), dataset (2 hops) →
{A1,A2,A3} × {compute, storage, dataset}
  - Start from memory: Include dataset (1 hop), storage (2 hops) →
{A3,A4,A5} × {storage, dataset, memory}
  - Overlap at A3: Two joint contentions with partial overlap
```

```
Problem sizes: 3×3=9 and 3×3=9 instead of 5×4=20. Roughly 2x speedup.


Pareto gap: Misses potential trades between A₁ and A₅ (not in same
contention).
  If A₁ values compute highly and A₅ values memory highly,
  and they could trade via intermediaries A₂,A₃,A₄,
  separate arbitrations won't find this chain trade.
  Empirical gap estimation: ~5-15% of potential welfare for k=2.
```

Policy 3: Resource Type Compatibility Matrix

This policy allows operators to specify that certain resource type pairs should never be jointly optimized, based on domain knowledge about temporal cycles, optimization incompatibilities, or operational constraints:

```
public class CompatibilityMatrixPolicy implements ContentionGroupingPolicy
{
    private final Set<Pair<ResourceType, ResourceType>> neverGroupPairs;

    @Override
    public Set<Contention> groupContentions(
            Map<ResourceType, SingleResourceContention> singleContentions,
            ImmutableGlobalState state) {

        // Start with aggressive grouping
        Set<Contention> aggressiveGroups =
            new
AggressiveGroupingPolicy().groupContentions(singleContentions, state);

        // Split groups that violate compatibility matrix
        Set<Contention> refined = new HashSet<>();
        for (Contention joint : aggressiveGroups) {
            if (containsIncompatiblePair(joint.getResourceTypes())) {
                // Split into compatible subgroups
                refined.addAll(splitByCompatibility(joint));
            } else {
                refined.add(joint);  // Keep as-is
            }
```

```
        }

        return refined;
    }

    private boolean containsIncompatiblePair(Set<ResourceType> types) {
        for (ResourceType t1 : types) {
            for (ResourceType t2 : types) {
                if (t1 != t2 && neverGroupPairs.contains(Pair.of(t1, t2)))
{
                    return true;
                }
            }
        }
        return false;
    }
}
```

Configuration:

```
arbitration:
  grouping_policy:
    type: "compatibility_matrix"
    parameters:
      never_group_pairs:
        - [compute_units, lab_time_slots]       # Fast vs slow cycles
        - [dataset_access, physical_robot_hours] # Info vs physical
        - [gpu_hours, collaboration_credits]     # Different time scales

      rationale:
        compute_lab: "Compute reallocates every 100ms, lab every 10min.
                      Mixing creates lowest-common-denominator problem
                      where fast resources wait for slow arbitration."
        dataset_robot: "Dataset access is informational (instant),
                        robot hours are physical (booking/scheduling).
                        Different optimization objectives."
```

This policy enables domain-specific optimization based on operator expertise. For example, an operator managing a mixed research facility might know that computational resources (virtualized, time-sliced, rapidly reallocated) should not be jointly optimized with physical resources (lab equipment with booking systems, slower reallocation cycles). By separating these, the platform can optimize each at appropriate time scales.

The tradeoff is that separation might miss beneficial trades: maybe an agent values lab time highly and would be willing to give up compute for better lab access, and another agent has opposite preferences. The joint optimization would discover this trade, but separated optimizations won't. The operator's expertise must weigh this potential loss against operational benefits.

Policy 4: Size-Bounded with Fallback

This policy imposes hard limits on joint contention size, using fallback mechanisms when contentions exceed bounds:

```
public class SizeBoundedPolicy implements ContentionGroupingPolicy {
    private final int maxAgents;
    private final int maxResources;
    private final int maxVariables;
    private final FallbackPolicy fallbackPolicy;

    @Override
    public Set<Contention> groupContentions(
            Map<ResourceType, SingleResourceContention> singleContentions,
            ImmutableGlobalState state) {

        // Start with aggressive grouping
        Set<Contention> aggressiveGroups =
            new
AggressiveGroupingPolicy().groupContentions(singleContentions, state);

        Set<Contention> bounded = new HashSet<>();

        for (Contention joint : aggressiveGroups) {
            int n = joint.getInvolvedAgents().size();
            int m = joint.getResourceTypes().size();
            int vars = n * m;

            if (n <= maxAgents && m <= maxResources && vars <=
maxVariables) {
```

```
                // Within bounds - use exact joint optimization
                bounded.add(joint);
            } else {
                // Exceeds bounds - apply fallback
                logger.warn("Joint contention exceeds size bounds " +
                           "(n={}, m={}, vars={}). Applying fallback: {}",
                           n, m, vars, fallbackPolicy.getName());

bounded.addAll(fallbackPolicy.handleOversizedContention(joint, state));
            }
        }

        return bounded;
    }
}
```

Configuration:

```
arbitration:
  grouping_policy:
    type: "size_bounded"
    parameters:
      max_agents: 50       # Never optimize more than 50 agents jointly
      max_resources: 20    # Never optimize more than 20 resource types
jointly
      max_variables: 500   # Never exceed 500 total variables (n×m)

      fallback_policy: "hierarchical"  # What to do when bounds exceeded
      # Options: hierarchical, approximate, sequential, sampling
```

Fallback Mechanisms:

When joint contention exceeds size bounds, fallback policies provide approximate allocations:

Hierarchical Arbitration:

```
1. Partition agents into clusters (e.g., 4 clusters of 15 agents each from
60-agent contention)
2. Within each cluster: Arbitrate using full joint optimization
3. Between clusters: Coordinate allocations (ensure total ≤ resource
limits)
```

```
4. Iterate: Adjust allocations to improve cross-cluster fairness


Approximation bound: Achieves ≥(1-ε) of optimal welfare where ε depends on
                     how well clusters partition (similar preferences
within cluster).
Phase 2 work: Prove bounds for different partitioning strategies.
```

Approximate Optimization:

```
1. Use gradient descent with fixed iteration budget (e.g., 100 iterations)
2. Initialize from previous solution (warm start) or heuristic
(proportional allocation)
3. Terminate after fixed time (500ms) regardless of convergence
4. Return best solution found


Approximation bound: Gradient descent converges at rate O(1/√K) where K is
iterations.
                     After 100 iterations, within ~10% of optimal for
well-conditioned problems.
Phase 2 work: Analyze convergence for our specific problem structure.
```

Sequential Arbitration:

```
1. Order resource types by contention severity (highest demand/supply ratio
first)
2. Arbitrate each resource type in sequence, considering previous
allocations
3. Earlier resources use standard Proportional Fairness
4. Later resources optimize subject to constraints from earlier allocations


Approximation bound: Sequential loses cross-resource trade opportunities.
                     Gap depends on preference correlation across
resources.
                     Worst case: ~30-50% gap if preferences highly
complementary.
                     Typical: ~10-20% gap for weakly correlated
preferences.
Phase 2 work: Characterize when sequential is acceptable vs. problematic.
```

Sampling:

```
1. Sample subset of agents (e.g., 30 agents from 100-agent contention)
```

```
2. Arbitrate full joint optimization for sampled subset
3. Extrapolate to full population (allocate to unsampled agents
proportionally)

Approximation bound: Depends on sample size and population heterogeneity.
                     For n_sample=30 from n_total=100: ~15% gap expected.
Phase 2 work: Prove bounds using sampling theory and mechanism properties.
```

Each fallback includes formal analysis of when it's appropriate and how much optimality is lost, enabling operators to make informed choices.

Policy 5: Parallel Safe (Automatic Optimization)

This policy automatically detects when contentions can be safely arbitrated in parallel (agent sets are disjoint, so no cross-resource trades possible) and exploits parallelism:

```
public class ParallelSafePolicy implements ContentionGroupingPolicy {
    @Override
    public Set<Contention> groupContentions(
            Map<ResourceType, SingleResourceContention> singleContentions,
            ImmutableGlobalState state) {

        // Build overlap graph
        Graph<SingleResourceContention> overlapGraph =
buildOverlapGraph(singleContentions);

        // Find connected components (as in aggressive policy)
        Set<Set<SingleResourceContention>> components =
            overlapGraph.findConnectedComponents();

        // Each disconnected component can be arbitrated in parallel
        // because they have disjoint agent sets (no possible
cross-resource trades)
        return components.stream()
            .map(this::createJointContention)
            .collect(Collectors.toSet());
    }

    @Override
    public double estimateParetoGap() {
```

```
        return 0.0;  // No gap when agent sets are truly disjoint
    }
}
```

Configuration:

```
arbitration:
  grouping_policy:
    type: "parallel_safe"
    # Automatically finds disjoint contentions and arbitrates in parallel
    # No operator configuration needed - policy is automatic
```

Example:

```
Single-resource contentions:
  Compute: Agents {A1, A2, A3}
  Storage: Agents {A4, A5, A6}
  Dataset: Agents {A7, A8}

Agent sets are completely disjoint (no overlap).
Overlap graph has three disconnected components.

Result: Three separate joint contentions (though "joint" is misnomer here -
        each is actually single-resource since agents within component
        don't overlap on multiple resource types):
  - Component 1: {A1,A2,A3} × {compute}
  - Component 2: {A4,A5,A6} × {storage}
  - Component 3: {A7,A8} × {dataset}

These can be arbitrated in parallel (submit to three executor threads,
solve simultaneously).

Speedup: ~3x if all problems are similar size (wall-clock time = max of the
three).

Pareto optimality: Still global optimal because no cross-component trades
exist.
  When agent sets are disjoint, impossible for trade across components to
  create Pareto improvement (agents in different components don't compete).
```

This policy provides "free" performance optimization when workloads naturally partition. If agents have specialized resource interests (ML agents want compute/GPU, robotics agents want lab/sensors, collaborative agents want workspace/collab), they form disjoint contentions that can parallelize. The policy automatically detects and exploits this structure.

Grouping Policy Evaluation Framework:

Phase 2 will include infrastructure for evaluating and comparing policies:

```
public interface GroupingPolicyMetrics {
    /**
     * Pareto gap: percentage of potential welfare improvements missed.
     * Computed by comparing achieved welfare to hypothetical optimal
     * from aggressive grouping.
     */
    double getParetoGap();

    /**
     * Latency improvement: reduction in arbitration time from policy.
     * Negative if policy actually increases latency (shouldn't happen,
     * but worth measuring).
     */
    double getLatencyImprovement();

    /**
     * Throughput: sustained arbitrations per second.
     */
    double getThroughput();

    /**
     * Policy-specific metadata.
     */
    Map<String, Object> getMetadata();
}

public interface AdaptiveGroupingPolicy extends ContentionGroupingPolicy {
    /**
     * Learn which policy works best for current workload.
```

```
     * Observes outcomes and adapts strategy.
     */
    void observeOutcome(
        Contention grouped,
        AllocationResult result,
        GroupingPolicyMetrics metrics
    );
}
```

An adaptive policy might start with aggressive grouping, measure the Pareto gap and latency, then switch to k=2 transitive depth if the gap is small (<5%) but latency improves significantly (>2x). Over time, it learns which policy is best for the current workload's characteristics.

Comparison Table for Operator Guidance:

Table 4. Comparison of Contention Grouping Policies

Policy	Global Pareto Optimal	Latency Characteristics	Complexity	Operator Configuration
Aggressive (M1)	Yes (Property 2.3)	O((n×m)^2.5) worst case	Medium	None (automatic)
Transitive k=2	~90-95% optimal	O(((n/3)×m)^2.5) typical	Low	Set k parameter
Compatibility Matrix	Depends on matrix	Variable by separation	Low	Specify never-group pairs
Size Bounded	Via fallback	Capped at threshold	Low	Set bounds + fallback
Parallel Safe	Yes (when disjoint)	O(max_component^2.5)	Low	None (automatic)

This table helps operators choose policies based on their priorities: if global optimality is critical (research applications where fairness matters most), use aggressive. If latency is critical (production applications with tight SLAs), use size-bounded with approximate fallback. If workload naturally partitions (specialized agents), parallel safe provides best of both.

11.5 Sophisticated Priority Economy Extensions

The Milestone 1 Priority Economy is elegantly simple: earnings are linear in TimeRemaining, demand multipliers are instantaneous ratios, and currency is permanently destroyed when

burned. Phase 2 can add sophistication while maintaining the closed-loop structure that creates self-regulating equilibria.

Time-Decay Earnings (Exponential Incentive):

Replace linear TimeRemaining with exponential decay:

```
earnings_j = quantity_j × exp(TimeRemaining_j / LeaseLength_j) ×
DemandMultiplier_j
```

This creates much stronger early-release incentive. An agent with 10-timestep lease who releases at t=2 (TimeRemaining=8) earns exp(0.8) ≈ 2.2× baseline. At t=8 (TimeRemaining=2), they earn exp(0.2) ≈ 1.2× baseline. The exponential amplifies differences, potentially accelerating resource return rates and improving system liquidity.

Configuration:

```
priority_economy:
  earning_model: "exponential_decay"
  parameters:
    decay_rate: 1.0  # Controls exponential steepness
    # Higher values = stronger early-release incentive
```

Phase 2 theoretical work: Analyze equilibrium dynamics under exponential decay. Does system reach stable equilibria? Do agents adapt strategies? Is there an optimal decay rate that maximizes social welfare?

Smoothed Demand Multipliers:

Replace instantaneous demand/supply with exponential moving average:

```
DemandMultiplier_j(t) = α · (CurrentDemand_j(t) / CurrentSupply_j(t))
                      + (1-α) · DemandMultiplier_j(t-1)
```

Smoothing parameter $\alpha \in (0,1)$ controls responsiveness: $\alpha=1.0$ is instantaneous (Milestone 1), $\alpha=0.3$ smooths over ~3 time periods, $\alpha=0.1$ smooths over ~10 periods. Smoothing prevents multipliers from spiking on transient demand fluctuations, creating more stable earning expectations.

Agents can then make medium-term strategic decisions: "demand multiplier for compute has been elevated at 2.5× for last 20 timesteps, likely to persist, I should release compute resources to earn premium rather than holding."

Currency Recycling vs. Permanent Destruction:

Milestone 1 permanently destroys burned currency (removed from system forever). Phase 2 might experiment with recycling:

```
When agent i burns C currency:
  1. Decrement balance_i by C (as in Milestone 1)
  2. Instead of destruction, redistribute C to all other agents
     proportionally to their current balances:

     For each agent j ≠ i:
       minting_j = C × (balance_j / Σ_k≠i balance_k)
```

This keeps total currency supply constant (InitialEndowments never changes), redistributing from spenders to holders. Might create more stable dynamics (no secular deflation from net burning > net earning) but requires analysis:

- Does redistribution break incentive properties? (Agents might burn knowing they'll get some back through redistribution)
- Does it create wealth stratification? (Rich get richer through redistribution)
- Does it affect equilibria qualitatively? (Time to equilibrium, balance distributions)

Phase 2 empirical work: Implement both destruction and recycling, measure dynamics, compare welfare outcomes and fairness metrics. Choose the model that better satisfies our objectives (efficiency, fairness, stability).

11.6 Security Hardening and Byzantine Tolerance

Milestone 1 assumes non-adversarial agents: they might compete strategically, but they don't actively attempt to break the platform or harm others maliciously. Phase 2 relaxes this assumption, adding mechanisms to detect and handle Byzantine behavior (arbitrary malicious actions including lying about utilities, coordinating attacks, or attempting to disrupt platform operation).

Adversarial Behaviors and Detection:

Byzantine agents might:

- **Report inconsistent utilities:** `evaluateUtility()` returns values inconsistent with declared preferences (attempting to game the optimization)
- **Refuse to release resources:** Hold resources indefinitely despite lease expirations (griefing other agents)
- **Timeout attacks:** Behavioral methods intentionally take maximum time (200ms) to slow platform
- **Spam requests:** Submit many rapid requests to trigger large joint contentions and exhaust computational budget

- **Coalition attacks:** Multiple agents coordinate to exploit mechanism properties we haven't proven (gaps in Theorem 3's heuristic argument)

Detection Interface:

```
public interface ByzantineDetector {
    /**
     * Assess single agent's behavior for Byzantine patterns.
     *
     * @param agent Agent to assess
     * @param state Current state
     * @param recentEvents Recent events involving this agent
     * @return Suspicion score ∈ [0,1], >0.7 suggests likely Byzantine
     */
    double assessAgent(
        AgentId agent,
        GlobalState state,
        List<Event> recentEvents
    );

    /**
     * Assess potential coalition based on coordinated behavior.
     *
     * @param suspectedCoalition Agents suspected of coordinating
     * @param state Current state
     * @return Collusion confidence ∈ [0,1]
     */
    double assessCollusion(
        Set<AgentId> suspectedCoalition,
        GlobalState state
    );

    /**
     * Recommend action for detected Byzantine agent.
     */
    ResponseAction recommendAction(AgentId agent, double suspicionScore);
}
```

```
public enum ResponseAction {
    MONITOR,        // Suspicion low, just watch closely
    RATE_LIMIT,     // Suspicion moderate, limit request frequency
    QUARANTINE,     // Suspicion high, stop agent and release resources
    BAN             // Confirmed malicious, permanently exclude
}
```

Detection heuristics might include:

- **Utility consistency:** Compare `evaluateUtility()` outputs to expected values from declared preferences across multiple allocations
- **Release compliance:** Track whether agents release resources promptly or always wait for forced expiration
- **Request patterns:** Detect unusual patterns (suddenly requesting many types, extreme quantity changes)
- **Coalition correlation:** Look for suspiciously coordinated request timing or preference alignment

These heuristics won't catch all Byzantine behavior (sophisticated adversaries can evade pattern detection), but they provide defense-in-depth when combined with the structural collusion resistance from Theorem 3.

Quarantine and Recovery:

When Byzantine behavior is detected with high confidence:

1. **Immediate quarantine:** Stop calling agent's behavioral methods, preventing further malicious actions
2. **Resource reclamation:** Force release of all held resources back to pool
3. **Currency freeze:** Prevent agent from earning or burning currency
4. **Event logging:** Record all events involving agent for forensic analysis
5. **Admin notification:** Alert operators for manual investigation
6. **Potential recovery:** If investigation determines false positive, agent can be restored

The quarantine is reversible but errs on the side of caution—better to temporarily restrict an honest agent (if detection made a mistake) than to allow a Byzantine agent to continue causing harm.

11.7 Service Resource Integration and Compositional Safety

Phase 2 extends the platform to coordinate access to specialized AI services—narrow, composable capabilities that agents invoke to accomplish complex tasks. This extension realizes a vision akin to Drexler's Comprehensive AI Services (CAIS): rather than monolithic general-purpose AI, agents compose workflows from specialized services like semantic paper search, molecular property prediction, experimental protocol generation, or hypothesis consistency checking. The platform treats these services as resource types, applying the same

fairness guarantees and Priority Economy dynamics that govern computational and physical resources.

Services as Resource Types

Service integration introduces a fifth resource category alongside computational, informational, physical, and collaborative resources. Services are organized into five subcategories based on their function: Perception services extract structured information from complex inputs (scientific figure data extraction, argument structure extraction), Analysis services perform bounded reasoning (hypothesis consistency checking, experimental outcome prediction), Generation services create structured outputs (protocol generation, literature review synthesis), Knowledge services access specialized information (semantic scholar search via dense retrieval, molecular property prediction), and Validation services check correctness (formal proof verification, citation integrity checking). Each service has explicit input/output schemas that constrain its domain (molecular property prediction accepts SMILES strings, returns structured property values), maintaining specialization through architectural boundaries.

From the mechanism's perspective, services are simply additional resource types. An agent requesting `{compute_units:40, semantic_scholar_search_credits:10, molecular_property_prediction_credits:100}` triggers contention detection that groups computational and service contentions into joint optimizations when agent sets overlap, exactly as for traditional resources. The arbitrator optimizes over the full allocation matrix including service types, discovering cross-resource trades: perhaps Agent A values compute highly while Agent B values molecular prediction highly, and the joint optimization allocates accordingly. The Priority Economy treats service credits identically—agents earn currency by releasing unused service credits early (credits × TimeRemaining × DemandMultiplier), burn currency for priority across all contested resource types (computational and service), and reach equilibria where earning rates balance spending rates.

New Infrastructure: Execution and Composition Monitoring

While the core coordination mechanism extends naturally to services, new infrastructure is required for execution and safety. The **ServiceRegistry** manages service backends (local ML models, external APIs, mock implementations for testing), providing discovery (agents query available services by category/capability), routing (platform selects appropriate backend based on load and health), and health monitoring (track backend availability, latency, throughput). The **ServiceInvocationManager** handles the lifecycle of service calls: accepting invocation requests from agents, debiting allocated credits, executing backend calls (potentially asynchronously for long-running services like experimental design optimization), tracking actual costs versus estimates (services may consume more or fewer credits than estimated depending on input complexity), and reconciling credits after completion. This manager integrates with the ResourceLedger to enforce credit conservation and with the Priority Economy to enable earning from unused service credits.

The **ServiceCompositionAnalyzer** provides safety monitoring for service chains. As agents invoke sequences of services—perhaps semantic search followed by argument extraction, molecular prediction, experimental design optimization, protocol generation, and consistency checking—the analyzer builds dependency graphs showing which services feed into which, measures composition depth (number of services in chain), and detects concerning patterns. Patterns warranting investigation include excessive depth (chains exceeding 10-15 services,

where capability might accumulate unpredictably), dangerous combinations (certain service pairs that together suggest capability-seeking behavior, like code generation combined with external network access, though this would require external services to exist in the first place), rapid temporal acceleration (agent suddenly invoking many more services than historically), or anomalous diversity (agent exploring many unrelated service types rather than specializing). The analyzer alerts operators to concerning patterns for manual review rather than enforcing hard limits, enabling empirical safety through monitoring while maintaining agent autonomy.

Compositional Safety Through Information Bottlenecks

A key safety hypothesis—currently aspirational rather than formally proven—is that service composition cannot create general intelligence because services have information bottlenecks at boundaries. Each service boundary involves information loss: semantic search returns paper references but loses full text, structured outputs compress away nuance, domain-specific training prevents generalization across domains (chemistry services can't process economics). If information retention factor α per service boundary is less than 1 (perhaps $\alpha \approx 0.7$, representing 30% information loss), then capability saturates as composition depth increases, with diminishing returns beyond depth 10-15 regardless of further composition. This is speculative—narrow services might still distill substantial information and utility—but provides a conceptual framework for Phase 2's empirical investigation. The ServiceCompositionAnalyzer will measure information retention across service chains, test whether capability saturates empirically, and build evidence for or refute the compositional safety hypothesis through systematic experimentation. The monitoring infrastructure enables gathering this evidence regardless of whether formal bounds ultimately hold, which will also be informative for a more theoretical study of the matter.

Architectural Integration Points

Service integration touches several existing components minimally. The **ResourceLedger** extends to track service credit balances per agent alongside computational resources, with conservation invariants applying uniformly. The **AgentBehavior** interface adds service invocation methods (invokeService returning results or futures, checkServiceCredits querying available quotas), with error handling for service-specific failures (InsufficientServiceCreditsException, ServiceTimeoutException, BackendUnavailableException). The **SafetyMonitor** applies existing invariants (conservation, non-negativity) to service resources and adds new checks (service credit allocations don't exceed agent balances, composition depth doesn't exceed configured soft limits if enforced). The **ContentionDetector** detects service contentions identically to computational contentions, and aggressive grouping merges them into joint contentions when agent sets overlap, enabling global Pareto optimality across mixed resource types.

Implementation proceeds in stages: initial Phase 2 uses mock service backends (simple heuristics like keyword-based search, random molecular properties within typical ranges) to validate infrastructure without requiring sophisticated models, then replaces critical services with real implementations (actual semantic search via embeddings, actual molecular prediction via trained models), and finally deploys production-quality backends with robust error handling, load balancing, and external API integration where appropriate. This staged approach validates mechanism properties (fairness, efficiency, compositional safety) before investing in full service implementations, following the same philosophy as Milestone 1's deliberate constraints: establish foundations with provable properties, then add sophistication systematically.

12. Deployment and Configuration

This section specifies how the platform is deployed and configured for operation, focusing on the single-JVM deployment model for Milestone 1. The configuration formats and deployment procedures are designed to be simple enough for immediate use while establishing patterns that generalize to distributed deployment later.

12.1 Platform Configuration Schema

Platform configuration uses YAML for human readability and supports hierarchical structure with inline documentation. The configuration file specifies resource pool quantities, execution parameters, arbitration settings including joint contention handling, Priority Economy parameters, and safety constraints.

Complete Configuration for Milestone 1:

```
platform:
  version: "0.1"
  phase: "milestone_1"
  deployment_mode: "single_jvm"

execution:
  thread_pool_size: 8              # Agent executor thread pool
  agent_timeout_ms: 200            # Behavioral method timeout
  embargo_window_ms: 100           # Request batching window
  contention_check_interval_ms: 50 # Detection frequency
  main_loop_poll_ms: 10            # Event loop polling interval

resources:
  pool:
    # Computational resources
    compute_units: 1000
    memory_units: 500
    storage_blocks: 2000
    gpu_hours: 100

    # Informational resources
    dataset_access: 10
    model_inference_credits: 5000
    api_call_quota: 1000
```

```
    knowledge_base_tokens: 100000

    # Physical resources (mocked for Milestone 1)
    lab_time_slots: 50
    sensor_access_windows: 60
    physical_robot_hours: 30
    3d_printer_reservations: 20

    # Collaborative resources
    collaboration_credits: 80
    shared_workspace_hours: 50
    communication_bandwidth: 10000

  lease_duration:
    default_timesteps: 10
    # Per-resource-type overrides:
    compute_units: 20               # Longer for compute-intensive tasks
    gpu_hours: 30                   # Even longer for GPU tasks
    lab_time_slots: 5               # Shorter for physical equipment
    physical_robot_hours: 5         # Physical resources have booking
constraints

arbitration:
  mechanism: "weighted_proportional_fairness"
  solver: "clarabel"
  solver_backend: "native_rust"

  base_priority_weight: 10.0
  computation_timeout_ms: 1000     # Fail if arbitration exceeds 1 second
  convergence_tolerance: 1.0e-6
  max_iterations: 100

  integer_handling:
    method: "round_then_adjust"
    max_adjustment_iterations: 10
    feasibility_tolerance: 5        # Allow 5 units excess during rounding

  contention_threshold: 1.0        # Trigger when demand ≥ supply
```

```
  # Joint contention settings
  joint_contention:
    grouping_policy: "aggressive"  # Milestone 1 unconditional
    # Phase 2 options: "transitive_depth_limit", "compatibility_matrix",
    #                  "size_bounded", "parallel_safe"

    # Parameters for Phase 2 policies (inactive in Milestone 1):
    policy_parameters:
      # For transitive_depth_limit:
      max_hops: 2

      # For size_bounded:
      max_agents: 50
      max_resources: 20
      max_variables: 500
      fallback_policy: "hierarchical"

      # For compatibility_matrix:
      never_group_pairs: []  # Empty for Milestone 1

    atomicity_enforcement: true    # Enforce Property 6.3
    transaction_timeout_ms: 5000   # Max time for full joint transaction

currency:
  initial_endowment: 1000
  minimum_balance: -100            # Bounded debt
  destruction_model: "permanent"   # Milestone 1: burned currency destroyed
                                   # Phase 2: "redistributed"?

priority_economy:
  earning_model: "time_scarcity_weighted"
  time_value_scaling: 1.0          # Linear TimeRemaining (Milestone 1)
                                   # Phase 2: >1.0 for exponential
  demand_multiplier_calculation: "simple_ratio"  # demand / supply
  demand_multiplier_high_value: 10.0            # When supply = 0
  demand_multiplier_smoothing: 0.0              # No smoothing (Milestone
1)
```

```
                                                    # Phase 2: 0.3 for EMA

safety:
  invariants:
    - "resource_conservation"
    - "non_negativity"
    - "commitment_satisfaction"
    - "currency_accounting"

  enforce_before_transitions: true
  periodic_validation_interval_ms: 5000  # Full validation every 5 seconds

  sandbox:
    agent_timeout_ms: 200
    max_memory_mb: 100               # Soft limit, monitored
    classloader_isolation: false     # Phase 2: true

logging:
  level: "INFO"
  audit_trail: true
  event_log_path: "./logs/events.jsonl"
  state_snapshots: true
  snapshot_interval_ms: 60000

monitoring:
  metrics_enabled: true
  metrics_port: 9090
  metrics_include:
    - "arbitration_latency"
    - "agent_timeout_rate"
    - "resource_utilization"
    - "currency_balance_distribution"
    - "contention_frequency"
    - "joint_contention_size_distribution"
    - "grouping_policy_effectiveness"
    - "atomicity_rollback_rate"
    - "cross_resource_trade_frequency"
```

Configuration Validation:

On platform startup, configuration is validated through three-tier validation:

Tier 1: Schema validation
- All required fields present (platform.version, resources.pool, arbitration.mechanism)
- Types correct (thread_pool_size is integer, base_priority_weight is double)
- Enums valid (mechanism ∈ {weighted_proportional_fairness}, grouping_policy ∈ {aggressive, ...})
- Ranges sensible (timeouts > 0, currency parameters ≥ 0)

Tier 2: Semantic validation
- Resource quantities positive and reasonable (pool quantities > 0, not excessively large)
- Timeout values reasonable (agent_timeout large enough for computation but not infinite)
- Currency parameters coherent (initial_endowment > 0, minimum_balance < 0 for debt)
- Lease durations appropriate (default_timesteps ≥ 1, per-type overrides ≥ 1)

Tier 3: Consistency validation
- Resource types in lease_duration overrides exist in pool
- Grouping policy parameters consistent with policy type (transitive_depth_limit requires max_hops, size_bounded requires bounds)
- Never-group pairs in compatibility matrix reference valid resource types
- Monitoring metrics requested are supported by platform

Failed validation prevents startup with detailed error messages explaining violations and suggesting corrections.

Configuration Profiles:

To simplify testing and deployment, the platform includes predefined configuration profiles:

minimal.yaml: 3 agents, 3 resource types (compute, storage, dataset), 100 units each. For unit testing and quick validation.

standard.yaml: 20 agents, 10 resource types spanning all four categories, 1000 units each. Milestone 1 target scale for full validation.

stress.yaml: 50 agents, 20 resource types, 2000 units each. For performance testing and scalability exploration. Will likely trigger Phase 2's need for grouping policy optimizations.

adversarial.yaml: 20 agents with deliberately conflicting preferences (all want same resources), low resource quantities (high contention), aggressive currency spending. For testing collusion resistance and fairness under scarcity.

Profiles enable quick deployment: `java -jar platform.jar --config profiles/standard.yaml --agents agents/*.yaml`

12.2 Agent Configuration Loading and Validation

Agent configurations follow the schema specified in the theory document's Appendix D. The architecture's AgentConfigurationLoader component handles parsing, validation, and instantiation, ensuring only well-formed agents enter the platform.

Loading Process:

```
public class AgentConfigurationLoader {

    public static AgentConfiguration loadFrom(Path jsonPath, Path
behaviorJar)
            throws ConfigurationException {


        // Step 1: Parse JSON

        JSONObject json = parseJSON(jsonPath);


        // Step 2: Validate against schema (Appendix D from theory doc)

        SchemaValidator.validate(json, AGENT_SCHEMA);


        // Step 3: Extract declarative components

        AgentId agentId = AgentId.of(json.getString("agent_id"));

        PreferenceFunction prefs =
parsePreferences(json.getJSONObject("preferences"));

        List<Goal> goals = parseGoals(json.getJSONArray("goals"));

        ResourceBundle initial =
parseInitialResources(json.getJSONObject("resources"));

        Set<ResourceType> interests =
parseResourceInterests(json.getJSONArray("resource_interests"));


        // Step 4: Load and instantiate behavior class

        String behaviorClassName = json.getString("behavior_class");

        ClassLoader loader = new URLClassLoader(new
URL[]{behaviorJar.toUri().toURL()});

        Class<? extends AgentBehavior> behaviorClass =

            (Class<? extends AgentBehavior>)
loader.loadClass(behaviorClassName);


        // Step 5: Validate behavior class implements interface correctly

        if (!AgentBehavior.class.isAssignableFrom(behaviorClass)) {

            throw new ConfigurationException(

                "Behavior class " + behaviorClassName +

                " does not implement AgentBehavior interface"
```

```
            );
        }

        // Step 6: Semantic validation
        validatePreferenceWeights(prefs);  // Sum to 1.0, all positive
        validateGoalsAchievable(goals, platformResourcePool);  //
Thresholds ≤ pool
        validateConsistency(prefs, goals, interests);  // Cross-field
coherence

        // Step 7: Package configuration
        return new AgentConfiguration(
            agentId, prefs, goals, initial, interests, behaviorClass
        );
    }
}
```

The loader performs extensive validation to catch configuration errors before the agent executes. Catching errors at load time (configuration validation) is far preferable to catching at runtime (execution failure).

Validation Rules (from Theory Appendix D):

Preference weights:
- Must sum to 1.0 within tolerance ($|\Sigma_i w_i - 1.0| < 10^{-6}$)
- All weights strictly positive ($w_i > 0$, no zero weights)
- Resource types referenced exist in platform pool
- Weights and resource interests are consistent (all weighted resources appear in interests)

Goals:
- Thresholds are achievable (threshold ≤ pool quantity for that resource type)
- Deadlines are positive (by_timestep > 0 for achieve_threshold goals)
- Resource types exist in platform
- No contradictory goals (maintain X ≥ 50 and maintain X ≤ 30 would be contradictory)

Initial resources:
- All quantities non-negative
- Total initial resources across all agents doesn't exceed pool (platform must have enough to endow all agents)

Behavior class:
- Class exists and is loadable from provided JAR
- Implements AgentBehavior interface with all required methods
- Has public no-argument constructor (for instantiation)

Violations of these rules cause ConfigurationException with detailed messages explaining what's wrong.

12.3 Deployment Procedures and Operational Guidelines

Single-JVM Deployment (Milestone 1):

The platform is packaged as a single executable JAR with all dependencies embedded (Clarabel4j native library, persistent data structure library, logging framework):

```
# Build platform
mvn clean package

# Produces:
target/arb-platform-0.1.jar          # Platform JAR (~15-25 MB)
target/arb-agents-examples-0.1.jar   # Example agent behaviors (~3-5 MB)

# Deploy
mkdir deployment
cp target/arb-platform-0.1.jar deployment/
cp -r config deployment/
cp -r agents deployment/

# Run
cd deployment
java -Xmx2G -Xss1M -XX:+UseG1GC \
  -jar arb-platform-0.1.jar \
  --config config/platform-standard.yaml \
  --agents agents/*.yaml
```

JVM Tuning Recommendations:

-Xmx2G: Maximum heap 2GB is sufficient for 20 agents with full event history and state snapshots. For larger agent populations or extended runtime (days), increase to 4-8GB.

-Xss1M: Thread stack size 1MB accommodates deep recursion in agent behavioral code (some decision algorithms might use recursive tree search). Default 512KB sometimes insufficient.

-XX:+UseG1GC: G1 garbage collector provides low-latency pauses (5-20ms typically), important for responsive coordination platform. Alternative: ZGC for even lower latency (<10ms) if available (Java 15+).

-XX:+UseStringDeduplication: String deduplication reduces memory from repeated string instances (agent IDs, resource type identifiers appear frequently). Can save 10-20% heap.

Platform Startup Sequence:

When platform starts:

1. **Load configuration** from specified YAML file, validate schema and semantics
2. **Initialize resource pool** with configured quantities for all resource types
3. **Load agent configurations** from specified directory, validate each against Appendix D schema
4. **Instantiate components** (EventBus, ContentionDetector, Arbitrator, ResourceManager, etc.) with configuration parameters
5. **Register agents** with AgentExecutionManager, instantiating behavior classes
6. **Create initial state** with resource pool, agent initial endowments, zero event history
7. **Start monitoring threads** (contention detection, embargo processing)
8. **Enter main event loop** and begin processing events

Startup typically completes in 1-5 seconds for standard configuration (20 agents, 10 resource types). Most time is spent loading and validating agent configurations (parsing JSON, loading behavior JARs, instantiating classes).

Monitoring and Observability:

The platform exposes HTTP/WebSocket endpoints for operational monitoring:

Metrics endpoint: `GET http://localhost:9090/metrics` returns Prometheus-compatible metrics:

```
# Arbitration latency histogram
arbitration_latency_ms_bucket{le="50"} 245
arbitration_latency_ms_bucket{le="100"} 678
arbitration_latency_ms_bucket{le="200"} 892
arbitration_latency_ms_bucket{le="500"} 987
arbitration_latency_ms_bucket{le="+Inf"} 1000

# Joint contention size distribution
joint_contention_size_bucket{le="10"} 150   # Small joints (≤10 variables)
joint_contention_size_bucket{le="50"} 420   # Medium (≤50 variables)
joint_contention_size_bucket{le="100"} 580  # Large (≤100 variables)
joint_contention_size_bucket{le="200"} 650  # Very large (≤200 variables)

# Resource utilization per type
```

```
resource_utilization{type="compute_units"} 0.87
resource_utilization{type="storage_blocks"} 0.62
resource_utilization{type="dataset_access"} 0.95  # High utilization
resource_utilization{type="lab_time_slots"} 0.34
# ... (per type)

# Currency balance distribution
currency_balance_percentile{p="50"} 850    # Median balance
currency_balance_percentile{p="95"} 2400   # High balances
currency_balance_percentile{p="5"} 120     # Low balances

# Cross-resource trade frequency
cross_resource_trades_found{from="compute_units",to="storage_blocks"} 47
# How often joint optimizations found beneficial trades between resource
types
```

Event stream endpoint: `ws://localhost:9091/events` provides real-time WebSocket stream of events in JSON format for live monitoring dashboards or external analysis tools.

State snapshot endpoint: `GET http://localhost:9090/state` returns current global state as JSON (potentially large, 10-50MB for full state) for debugging or forensic analysis.

Health endpoint: `GET http://localhost:9090/health` returns platform health status:

```
{
  "status": "healthy",
  "version": "0.1",
  "uptime_seconds": 3672,
  "current_state_version": 8234,
  "active_agents": 18,
  "pending_contentions": 2,
  "recent_arbitration_latency_ms": 156,
  "lock_contention_rate": 0.03,
  "invariant_violations": 0
}
```

These endpoints enable integration with standard monitoring tools (Prometheus + Grafana for metrics/dashboards, external log aggregators for event streams, health check monitors for alerting).

12.4 Operational Considerations and Troubleshooting

Common Operational Scenarios:

High arbitration latency: If 95th percentile arbitration latency exceeds 500ms consistently, investigate:

- Are joint contentions larger than expected? (Check joint_contention_size_distribution metric)
- Is grouping policy creating giant joints? (All agents requesting one universal resource like compute)
- Are agent preferences causing poor conditioning? (Extreme weight ratios across resource types)

Solutions:

- Reduce contention threshold (trigger arbitration earlier, before contention becomes severe)
- Phase 2: Enable size-bounded policy with fallback to limit worst-case problem size
- Add more resources to pool (reduce demand/supply ratio, making contentions less severe)

Agent timeouts: If agent_timeout_rate metric shows frequent timeouts (>5% of calls), investigate:

- Which agents timing out? (Check per-agent timeout counters)
- Which methods timing out? (computeResourceNeeds vs. decidePriorityBudget vs. adaptToPartialAllocation)
- Is timeout duration too short? (200ms might be insufficient for complex agent logic)

Solutions:

- Increase timeout to 300-500ms (tradeoff: slower response when agents do timeout)
- Profile misbehaving agent code (is it genuinely complex or inefficient?)
- Rate-limit problematic agents (if timeouts are frequent, agent might be misbehaving)

Resource exhaustion across multiple types: If pool quantities drop to zero for several resource types simultaneously (heavy joint contentions allocating everything), agents will queue waiting for releases. Monitor:

- How long until resources released? (Track lease expirations)
- Are agents releasing promptly? (TimeRemaining in release events—should be >50% of lease duration)
- Is demand multiplier incentivizing release? (Should be elevated when pool is depleted)

Solutions:

- Increase pool quantities (add more physical resources or expand virtualized capacity)
- Encourage agents to release promptly (adjust time_value_scaling in Priority Economy)
- Phase 2: Add predictive arbitration to anticipate exhaustion before it occurs

Invariant violations: If SafetyMonitor detects violations (should be extremely rare), this indicates serious bugs:

- Resource conservation violation: Accounting error in transfer logic, likely in joint allocation enforcement

- Non-negativity violation: Arithmetic underflow or incorrect release handling
- Commitment satisfaction violation: Arbitrator allocated more than exists (bug in formulation)

Response:
- **Immediate:** Platform should halt (fail-safe) rather than continuing with corrupted state
- **Investigation:** Review event history leading to violation, identify which component introduced corruption
- **Fix:** Patch the bug, validate fix with comprehensive testing including joint contention scenarios
- **Recovery:** Restart platform from last valid state snapshot (if available), or from initial state (lose work but maintain correctness)

Invariant violations should never occur in correct implementations—their presence indicates fundamental bugs that must be fixed before deployment continues.

Monitoring Joint Contention Health:

Metrics enable monitoring joint contention behavior:

joint_contention_size_distribution: Histogram showing how large joint problems are becoming. If 95th percentile exceeds 150 variables, approaching Phase 2 threshold where performance optimizations become necessary.

atomicity_rollback_rate: How often do joint allocation transactions roll back? Should be <1% (rare). High rollback rate suggests bugs in arbitration logic producing infeasible solutions, or pathological contention patterns.

cross_resource_trade_frequency: How often do joint optimizations discover beneficial trades across resource types? High frequency (many trades found) validates that aggressive grouping provides significant welfare improvements. Low frequency (<5% of joint arbitrations find trades) suggests workload might tolerate selective grouping policies.

These metrics inform operational decisions and guide Phase 2 development priorities.

Acknowledgements

This document was developed through human-AI collaboration, with the vision, architectural decisions, and mechanism design provided by human authors (Daniele Palombi an expert in mechanism design and Richard Mallah an expert in AI architecture), with assistance on specification writing, code example generation, and phrasing by Anthropic's Claude (Sonnet 4.5), where the authors reviewed and edited all such contributions.

This work was made possible by a grant from Foresight Institute.

Appendices

Appendix A: Complete Interface Catalog

This appendix provides a comprehensive listing of all major interfaces in the platform architecture, organized by architectural layer. This serves as a quick reference for implementation teams and documents extension points for Phase 2. This appendix catalogs primarily Milestone 1 interfaces. Phase 2's service integration will add ServiceRegistry, ServiceInvocationManager, ServiceCompositionAnalyzer, and ServiceBackend interfaces, which are not included here as they remain conceptual.

Core Platform Interfaces:

```
// Main runtime orchestrator
public interface PlatformRuntime {
    void initialize(PlatformConfiguration config,
Collection<AgentConfiguration> agents);
    void run();
    void shutdown();
    long getCurrentStateVersion();
    ImmutableGlobalState getCurrentState();
}


// Event communication backbone
public interface EventBus {
    void publish(Event event);
    <T extends Event> void subscribe(Class<T> eventType, EventHandler<T>
handler);
    <T extends Event> void unsubscribe(Class<T> eventType, EventHandler<T>
handler);
    Event getNextEvent() throws InterruptedException;
}


// Request batching for fairness
public interface EmbargoQueue {
    void submit(ResourceRequestEvent request);
    void setEmbargoWindowMs(long windowMs);
```

```
    long getEmbargoWindowMs();
}
```

Coordination Logic Interfaces:

```
// Contention detection with joint grouping
public interface ContentionDetector {
    /**
     * Detect contentions and group via configured policy.
     * Milestone 1: Aggressive grouping always.
     * Phase 2: Pluggable grouping policies.
     */
    Set<Contention> detectAndGroup(
        ImmutableGlobalState state,
        Collection<AgentExecutionContext> agents
    );

    void setContentionThreshold(double threshold);
    GroupingStatistics getGroupingStats();
}

// Grouping policy abstraction (Phase 2)
public interface ContentionGroupingPolicy {
    Set<Contention> groupContentions(
        Map<ResourceType, SingleResourceContention> singleContentions,
        ImmutableGlobalState state
    );
    String getPolicyName();
    Map<String, Object> getParameters();
    double estimateParetoGap();
}

// Core arbitration mechanism (handles joint problems)
public interface ProportionalFairnessArbitrator {
    /**
     * Compute globally Pareto-optimal allocation.
     * Handles both single-resource (m=1) and joint (m>1) uniformly.
```

```
     */
    AllocationResult arbitrate(
        Contention contention,
        ImmutableGlobalState state
    ) throws ArbitrationException;

    /**
     * Estimate joint problem complexity (for monitoring).
     */
    default int estimateProblemSize(Contention contention) {
        return contention.getInvolvedAgents().size() *
               contention.getResourceTypes().size();
    }
}

// Priority Economy calculations
public interface PriorityWeightCalculator {
    Map<AgentId, Double> calculatePriorityWeights(
        Map<AgentId, BigDecimal> commitments,
        ImmutableGlobalState state
    ) throws InsufficientCurrencyException;

    /**
     * Calculate earnings for multi-resource release.
     * Sums earnings across all resource types in bundle.
     */
    BigDecimal calculateReleaseEarnings(
        ResourceBundle resources,
        Map<ResourceType, Instant> leaseExpirations,
        Instant currentTime,
        ImmutableGlobalState state
    );

    BigDecimal calculateDemandMultiplier(ResourceType type, GlobalState
state);
}

// Resource enforcement with atomicity
```

```
public interface ResourceManager {
    /**
     * Enforce allocation atomically across all resource types.
     * Uses TransactionManager for Property 6.3 atomicity.
     */
    GlobalState enforceAllocation(
        AllocationResult result,
        GlobalState currentState
    ) throws InsufficientResourcesException, TransactionException;

    /**
     * Process multi-resource release, calculate earnings, mint currency.
     */
    GlobalState releaseResources(
        AgentId agentId,
        ResourceBundle resources,
        GlobalState currentState
    );

    boolean isAllocationFeasible(AllocationResult result, GlobalState
state);
    Map<ResourceType, Instant> getLeaseExpirations(AgentId agent,
GlobalState state);
    GlobalState reclaimExpiredLeases(GlobalState state);
}

// Transaction management for atomicity
public interface TransactionManager {
    Transaction begin();
    GlobalState execute(Transaction txn, StateOperation operation) throws
TransactionException;
    GlobalState commit(Transaction txn, GlobalState finalState) throws
TransactionException;
    void rollback(Transaction txn, GlobalState originalState);
}

// Safety invariant checking
public interface SafetyMonitor {
    ValidationResult validateState(GlobalState state);
```

```
    boolean isSafeTransition(Event event, GlobalState currentState);
    void registerInvariant(SafetyInvariant invariant);
}

public interface SafetyInvariant {
    boolean holds(GlobalState state, Event event);
    String getDescription();
    String getInvariantId();
}

// Agent lifecycle management
public interface AgentExecutionManager {
    void registerAgent(AgentConfiguration config) throws
ConfigurationException;
    ResourceRequest executeComputeNeeds(AgentId agent, GlobalState state)
        throws AgentExecutionException;
    BigDecimal executePriorityBudget(AgentId agent, GlobalState state,
ResourceRequest need)
        throws AgentExecutionException;
    ExecutionPlan executeAdaptToPartial(AgentId agent, ResourceRequest
requested,
                                        ResourceAllocation received)
        throws AgentExecutionException;
    Collection<AgentExecutionContext> getActiveAgents();
    void notifyAllocation(AgentId agent, ResourceAllocation allocation);
}
```

Agent Interface:

```
public interface AgentBehavior {
    ResourceRequest computeResourceNeeds(
        ImmutableAgentState myState,
        ImmutableGlobalState worldState
    ) throws AgentException;

    ExecutionPlan adaptToPartialAllocation(
        ResourceRequest requested,
        ResourceAllocation received
```

```
    ) throws AgentException;

    @Nullable
    PredictiveResourceSignal predictFutureNeeds(
        ImmutableAgentState myState,
        int horizonTimesteps
    );

    double evaluateUtility(ImmutableObservableState state);

    BigDecimal decidePriorityBudget(
        ImmutableAgentState myState,
        ImmutableGlobalState worldState,
        ResourceRequest currentNeed
    );
}

public interface PreferenceFunction {
    double evaluate(ResourceBundle allocation);
    Map<ResourceType, Double> getWeights();

    // Phase 2 additions:
    PreferenceType getType();
    boolean isConvex();
}

public interface Goal {
    boolean isSatisfied(ImmutableAgentState state);
    String getDescription();
    GoalType getType();
}
```

Extension Interfaces (Phase 2, not implemented in Milestone 1):

```
// Learning and prediction
public interface LearningModule {
    void observe(Event event);
```

```
    @Nullable PredictedDemand predict(ResourceType type, int horizon,
GlobalState state);

    Set<PredictedJointContention> predictJointContentions(int horizon,
GlobalState state);

}


// Byzantine detection

public interface ByzantineDetector {

    double assessAgent(AgentId agent, GlobalState state, List<Event>
recentEvents);

    double assessCollusion(Set<AgentId> suspectedCoalition, GlobalState
state);

    ResponseAction recommendAction(AgentId agent, double suspicionScore);

}


// Distributed consensus (some later scope)

public interface ConsensusProtocol {

    <T> T propose(T value) throws ConsensusException;

    ClusterHealth getHealth();

}


// Alternative mechanisms (for research comparison)

public interface ArbitrationMechanism {

    AllocationResult arbitrate(Contention contention, GlobalState state);

    String getMechanismName();

    Set<MechanismProperty> getGuarantees();

}


// Grouping policy evaluation

public interface GroupingPolicyMetrics {

    double getParetoGap();

    double getLatencyImprovement();

    double getThroughput();

    Map<String, Object> getMetadata();

}


public interface AdaptiveGroupingPolicy extends ContentionGroupingPolicy {

    void observeOutcome(Contention grouped, AllocationResult result,

                        GroupingPolicyMetrics metrics);
```

```
}
```

These extension interfaces document Phase 2's architecture without implementing it, providing clear targets for future development.

Appendix B: Detailed Event Flow for Joint Contention

This appendix provides a comprehensive sequence diagram showing how joint contentions flow through the platform from initial detection through atomic enforcement.

This sequence demonstrates the complete flow for a joint contention involving three agents and two resource types, showing how aggressive grouping creates a unified optimization problem, how atomic enforcement ensures all-or-nothing allocation, and how agents subsequently work with and release the allocated resources.

Key Timing Characteristics:

- Embargo delay: 100ms (fixed window)
- Priority budget queries: 3 agents × 5ms = 15ms
- Priority weight calculation: <1ms
- Joint arbitration: 60-120ms (dominates)
- Transaction enforcement: 5-10ms
- Agent notifications: <1ms (async, non-blocking)
- Total latency: ~180-240ms from request submission to allocation enforcement

For larger joint contentions (n=20, m=10), the arbitration time increases to 200-500ms, but other components remain fast. The arbitration computation is the scalability bottleneck, which Phase 2's grouping policies address.

Appendix C: Worked Examples Demonstrating Joint Optimization

This appendix expands the examples from the theory document's Appendix C with implementation-oriented details and adds a new example demonstrating worst-case joint contention behavior. These examples serve as validation test cases for Milestone 2 implementation.

Example 4: Worst-Case Joint Contention

This example demonstrates platform behavior when aggressive grouping creates the largest possible joint contention: all agents competing for one universal resource, causing all resource contentions to merge into a single giant optimization problem.

Scenario Setup:

20 agents, 10 resource types. All agents request compute_units (universal contention), plus each agent requests 2-3 additional unique resource types (creating overlaps that force everything into one joint contention via aggressive grouping).

Agent Configuration Pattern:

```
Agents 1-5 (Compute + Storage focused):
  Preferences: w_compute=0.60, w_storage=0.30, w_dataset=0.10
  Requests: compute:50, storage:40, dataset:2

Agents 6-10 (Compute + Physical focused):
  Preferences: w_compute=0.50, w_lab=0.30, w_robot=0.20
  Requests: compute:50, lab:5, robot:3

Agents 11-15 (Compute + Collaborative focused):
  Preferences: w_compute=0.50, w_collab=0.35, w_workspace=0.15
  Requests: compute:50, collab:8, workspace:5

Agents 16-20 (Compute + Informational focused):
  Preferences: w_compute=0.40, w_inference=0.35, w_api=0.25
  Requests: compute:50, inference:50, api:40
```

Resource Pool:

```
compute_units: 800           # Universal contention (total demand = 1000 > 800)
storage_blocks: 400          # Partial contention (agents 1-5 only)
dataset_access: 20           # Partial contention (agents 1-5 only)
lab_time_slots: 50           # Partial contention (agents 6-10 only)
physical_robot_hours: 30     # Partial contention (agents 6-10 only)
collaboration_credits: 100   # Partial contention (agents 11-15 only)
shared_workspace_hours: 60   # Partial contention (agents 11-15 only)
model_inference_credits: 800 # Partial contention (agents 16-20 only)
api_call_quota: 400          # Partial contention (agents 16-20 only)
sensor_access_windows: 40    # Uncontested (no agent requests - included for completeness)
```

Contention Analysis:

Phase 1: Single-resource contention detection

- Compute: 20 agents requesting 50 each = 1000 > 800 (1.25x contention) ✓
- Storage: 5 agents requesting 40 each = 200 < 400 ✗ (no contention)
- Dataset: 5 agents requesting 2 each = 10 < 20 ✗
- Lab: 5 agents requesting 5 each = 25 < 50 ✗
- Robot: 5 agents requesting 3 each = 15 < 30 ✗
- Collab: 5 agents requesting 8 each = 40 < 100 ✗
- Workspace: 5 agents requesting 5 each = 25 < 60 ✗
- Inference: 5 agents requesting 50 each = 250 < 800 ✗
- API: 5 agents requesting 40 each = 200 < 400 ✗
- Sensor: 0 agents requesting ✗

Only compute_units is actually contended!

Phase 2: Aggressive grouping analysis

Build overlap graph:

- Compute contention node: agents $\{A_1,...,A_{20}\}$ (all 20 agents)
- Since compute contention includes all agents, it overlaps with every other potential contention (even though those aren't individually contended)
- Wait, but storage/dataset/lab/etc. aren't contended (demand < supply)
- **Therefore: Only compute contention is detected**
- **Result: Single-resource contention for compute with 20 agents, NOT a joint contention**

Correction to Scenario:

To create a true worst-case joint contention, we need multiple resource types to be contended:

Adjusted Resource Pool:

```
compute_units: 800           # 1000 demanded, 1.25x contention ✓
storage_blocks: 150          # 200 demanded (agents 1-5), 1.33x contention
✓
model_inference_credits: 200 # 250 demanded (agents 16-20), 1.25x
contention ✓
api_call_quota: 150          # 200 demanded (agents 16-20), 1.33x
contention ✓
# Keep others uncontended for clarity
```

Now the contention analysis:

- Compute: $\{A_1,...,A_{20}\}$ contended ✓
- Storage: $\{A_1,A_2,A_3,A_4,A_5\}$ contended ✓
- Inference: $\{A_{16},A_{17},A_{18},A_{19},A_{20}\}$ contended ✓
- API: $\{A_{16},A_{17},A_{18},A_{19},A_{20}\}$ contended ✓

Overlap analysis:

- Compute overlaps with storage at $\{A_1,...,A_5\}$
- Compute overlaps with inference at $\{A_{16},...,A_{20}\}$

- Inference overlaps with API at $\{A_{16},...,A_{20}\}$
- All contentions form one connected component via compute as hub

Result: ONE giant joint contention:
- Agents: All 20 agents $\{A_1,...,A_{20}\}$
- Resource types: {compute, storage, inference, api} (4 types)
- Problem size: 20 agents × 4 resources = 80 variables
- Plus 20 auxiliary variables for t_i
- Total: 100 variables for Clarabel

Joint Optimization Problem:

```
maximize: Σᵢ₌₁²⁰ cᵢ · log(Φᵢ(A))

where for agents 1-5:
  Φᵢ(A) = 0.60·aᵢ,compute + 0.30·aᵢ,storage + 0.10·0 (no dataset allocation
in this joint)

for agents 6-10:
  Φᵢ(A) = 0.50·aᵢ,compute + 0 + 0 (no lab/robot in this joint, just
compute)

for agents 11-15:
  Φᵢ(A) = 0.50·aᵢ,compute + 0 + 0 (no collab/workspace in this joint)

for agents 16-20:
  Φᵢ(A) = 0.40·aᵢ,compute + 0.35·aᵢ,inference + 0.25·aᵢ,api

subject to:
  Σᵢ₌₁²⁰ aᵢ,compute ≤ 800
  Σᵢ₌₁⁵ aᵢ,storage ≤ 150
  Σᵢ₌₁₆²⁰ aᵢ,inference ≤ 200
  Σᵢ₌₁₆²⁰ aᵢ,api ≤ 150
  [minimums and maximums for each agent-resource pair]
```

Expected Computational Performance:

Table 5. Expected Computational Performance for Worst-Case Scenario

Metric	Value	Interpretation
Variables	100	20 agents × 4 resources + 20 auxiliary
Exponential cones	20	One per agent
Linear constraints	~164	4 resource limits + ~160 min/max bounds
Expected solve time	150-300ms	Based on O((n×m)^2.5) with n=20, m=4
Expected iterations	25-40	Typical for well-conditioned problems
Memory usage	~40-60MB	KKT matrix and workspace

This is well within Milestone 1's computational budget (<500ms at 95th percentile).

Expected Allocation Pattern:

Agents 16-20 (who value inference and API highly, w=0.35+0.25=0.60 total for those two) should receive generous allocations of inference/API since only 5 agents compete for those types. They receive moderate compute (competing with all 20 agents).

Agents 1-5 (who value storage highly, w=0.30) should receive good storage allocations since only 5 agents compete. Moderate compute.

Agents 6-15 (who only requested compute in this joint, since their preferred resources aren't contended) compete only on compute dimension. They receive compute proportional to their priority weights and compute preferences.

Cross-Resource Trades Discovered:

The joint optimization can find trades like:

- Give more compute to agents 6-10 (who value it at w=0.50 and have no other contested resources), take compute from agents 16-20 (who value it at w=0.40)
- Give more inference/API to agents 16-20 (who value them at w=0.35+0.25), since they're the only requesters
- Give more storage to agents 1-5, since they're the only requesters

Separate arbitrations would allocate compute proportionally across all 20 agents without considering that agents 1-5 and 16-20 have valuable outside options (storage, inference/API) that agents 6-15 lack. The joint optimization discovers these complementarities.

Validation Metrics:

Table 6. Validation Metrics for Worst-Case Scenario

Metric	Target	Purpose
Solve time (median)	150-200ms	Validate O(n^2.5) complexity
Solve time (95th %ile)	<300ms	Ensure within performance budget
Iterations	25-40	Confirm typical convergence
Pareto improvements found	≥15	Validate joint optimization finds trades
Agents receiving allocations	20	All agents participate (starvation protection)
Resource types fully allocated	2-3 of 4	Some types exhausted, others have slack

Scenario Purpose:

This worst-case scenario validates several critical properties:

1. **Aggressive grouping is computationally feasible** at Milestone 1 scale (n=20) even when creating large joint contentions
2. **Joint optimization finds meaningful cross-resource trades** (measured by Pareto improvements vs. hypothetical separate arbitrations)
3. **Atomicity enforcement works** for allocations spanning many agents and resource types
4. **Starvation protection holds** even in worst-case grouping (all 20 agents receive allocations despite competing in one giant optimization)

If this scenario performs well (solve time <300ms, all agents satisfied, significant trades found), it validates that aggressive grouping is the right Milestone 1 default. If performance degrades (solve time >500ms, numerical instabilities), it informs Phase 2's grouping policy design.

Appendix D: Validation Scenario Architectural Support

The theory document's Appendix E specifies six validation scenarios for Milestone 2 empirical testing. This appendix explains how the architecture supports each scenario, identifying which components and interfaces are exercised.

Scenario 1: Complementary Preferences (Theory Appendix E.1)

Four agents with diverse preferences across multiple resource types demonstrate coordination value through Pareto improvements.

Architectural Support:

- **ContentionDetector:** Detects multiple resource contentions, applies aggressive grouping to merge into joint contention based on agent overlaps
- **ProportionalFairnessArbitrator:** Computes joint allocation across all contested resource types simultaneously, finding cross-resource trades
- **ResourceManager:** Enforces multi-resource allocation atomically via TransactionManager
- **PriorityWeightCalculator:** Computes priority weights from varying currency commitments, calculates earnings per resource type when agents release

Metrics Collected:
- Social welfare before/after arbitration (measures Pareto improvement)
- Individual utilities per agent (validates individual rationality)
- Resource utilization per type (confirms efficient use)
- Cross-resource trades found (counts how many trades joint optimization discovered)

Expected Outcome: Total welfare increases 40-60% compared to independent operation baseline, validating Theorem 1's Pareto optimality and Property 2.3's global optimality.

Scenario 2: Homogeneous Competition (Theory Appendix E.2)

Five agents with identical preferences (w_compute=1.0) compete for compute_units at 1.6x demand/supply, stressing partial allocation and priority weight effectiveness.

Architectural Support:

- **Arbitration with varied priority weights:** Agents burn 0, 5, 20, 30, 50 currency, creating weight range 10-60
- **Partial allocation:** Agents specify minimums, receive allocations between min and ideal, must adapt
- **Priority effectiveness measurement:** Correlation between priority weight and allocation size

Metrics Collected:
- Allocation size vs. priority weight (should be correlated but sublinear)
- Fairness metrics (Gini coefficient of utility distribution, max/min utility ratio)
- Minimum satisfaction rate (100% when feasible)
- Currency consumption patterns (which agents deplete fastest)

Expected Outcome: Agent burning 50 currency receives largest allocation (~60 units), agent burning 0 receives smallest but still meaningful (~15 units, well above minimum=10). Logarithmic barrier prevents monopolization. Gini coefficient ~0.20-0.30 (moderate inequality).

Scenario 3: Collusion Attempt (Theory Appendix E.3)

Two coalition agents coordinate off-platform to starve victim agent, testing Theorem 3's structural collusion resistance.

Architectural Support:

- **Logarithmic barrier tracking:** Monitor how often mechanism reallocates to victim when coalition tries to monopolize

- **Coalition utility measurement:** Compare coalition's combined utility to honest baseline
- **Victim protection verification:** Ensure victim always receives ≥ minimum despite attack

Metrics Collected:
- Victim utility over time (should remain above minimum threshold)
- Coalition combined utility vs. honest baseline (collusion should be unprofitable)
- Frequency of protective reallocation (platform actively defending victim)

Expected Outcome: Victim receives 25-30% of contested resources despite zero currency and 51:1 priority weight disadvantage. Coalition receives ≤100% of honest baseline utility (collusion is not profitable).

Scenario 4: Dynamic Priority Signaling (Theory Appendix E.4)

Three agents with different demand patterns (frequent small requests, infrequent large requests, moderate regular requests) validate Priority Economy equilibration.

Architectural Support:

- **Currency balance tracking:** Monitor balances over 200 timesteps
- **Earning calculation:** Compute earnings using TimeRemaining and DemandMultiplier per resource type
- **Equilibrium analysis:** Identify when balances stabilize

Metrics Collected:
- Currency balance trajectories over time
- Time to equilibrium (when does variance in balance stabilize?)
- Earning rates per agent (income from releases)
- Spending rates per agent (currency burned per arbitration)

Expected Outcome: Balances converge to agent-specific equilibria within 50-100 arbitrations. Frequent releaser (P_1) accumulates more currency than bursty specialist (P_2) despite P_2's larger individual releases, because P_1's higher frequency dominates.

Scenario 5: Semantic Divergence Measurement (Theory Appendix E.5)

Agents with intentional observable-semantic gaps validate Theorem 4's and Theorem 4.1's divergence bounds.

Architectural Support:

- **Post-hoc utility collection:** Query agents about true semantic utility achieved (requires agents to implement optional feedback method)
- **Divergence calculation:** Compare semantic utility to observable utility
- **Bound verification:** Check $|\Phi\_sem - \Phi\_obs| \leq 2 \cdot ||\Phi\_hidden||\_\infty$
- **Correlation estimation:** Compute ρ = correlation(Φ_hidden under different allocations) for Theorem 4.1

Metrics Collected:
- Observable utility (what platform computed)

- Semantic utility (what agent actually achieved, via feedback)
- Hidden utility magnitude estimates
- Correlation coefficients between hidden and observable

Expected Outcome: For well-configured agents (comprehensive observable features), divergence <10% of utility. For poorly-configured agents (sparse observables), divergence 20-50% but still within 2·||Φ_hidden||_∞ bound.

Scenario 6: Worst-Case Joint Contention (Theory Appendix E.6)

See Example 4 above for complete specification. This scenario validates that aggressive grouping remains computationally tractable even when creating large joint contentions.

Architectural Support:

- **Large joint optimization:** 20 agents × 4 contested resource types = 80 variables
- **Atomic enforcement:** All 80 allocations enforced together (Property 6.3)
- **Performance monitoring:** Solve time, iteration count, memory usage for large problems
- **Trade discovery:** Count cross-resource Pareto improvements found by joint optimization

Metrics Collected:
- Solve time (should be <500ms at 95th percentile for n=20, m=4)
- Problem size distribution (tracks how large joints become in practice)
- Pareto improvements from joint vs. hypothetical separate arbitrations
- Transaction success rate (atomic enforcement should succeed >99% of time)

Expected Outcome: Solve time 200-400ms median, <500ms at 95th percentile. Joint optimization finds 15-30 beneficial trades that separate arbitrations would miss. All 20 agents receive allocations, demonstrating starvation protection scales.

Implementation Strategy for Milestone 2:

Each scenario will be implemented as an executable test:

```
@Test
public void testScenario1_ComplementaryPreferences() {
    // Setup: Create 4 agents with configurations from theory Appendix E.1
    // Execute: Run platform for 100 timesteps
    // Validate: Check welfare improvement, individual rationality, resource utilization
    // Assert: Welfare improvement ≥ 40%, all agents ≥ 120% of independent baseline
}
```

Tests will produce detailed reports with metrics, visualizations of currency dynamics, allocation patterns over time, and comparisons to theoretical predictions. These reports serve both as validation evidence and as documentation of system behavior.

Appendix E: Glossary of Architectural Terms

Aggressive Grouping: The Milestone 1 grouping policy where contentions are merged into joint contentions whenever they share any agents, ensuring global Pareto optimality (Property 2.3). Contrast with selective grouping policies in Phase 2.

Agent: Configuration-seeded process executing within platform sandbox. Consists of declarative preferences and goals plus executable behavioral code implementing AgentBehavior interface.

AgentBehavior: Java interface defining the behavioral methods agents must implement (computeResourceNeeds, adaptToPartialAllocation, decidePriorityBudget, etc.).

AgentExecutionContext: Runtime container wrapping an agent's configuration, behavior instance, and execution state. Maintained by AgentExecutionManager.

Allocation: Assignment of resource quantities to agents. For joint contentions, allocation is a matrix spanning multiple agents and resource types.

AllocationResult: Data structure returned by arbitrator containing allocation matrix, currency burned per agent, objective value, and solver metadata.

Arbitration: Process of computing fair allocation when contention exists. Uses Weighted Proportional Fairness mechanism via convex optimization.

Arbitration Lock: Level 3 lock in hierarchy, held during arbitration computation. May be held for 200-500ms for large joint contentions.

Atomicity (Property 6.3): Guarantee that joint allocations are enforced all-or-nothing—either all resources for all agents allocated together, or none are, with clean rollback on failure.

BaseWeight: Minimum priority weight (10.0) ensuring all agents participate even with zero currency. Foundation of starvation protection.

Burning (currency): Permanently destroying currency to signal urgency and increase priority weight in arbitration. Contrasts with payment (transferring currency between agents).

Clarabel: Rust-based conic optimization solver specializing in exponential cone programs. Used via Clarabel4j Java bindings for computing Proportional Fairness allocations.

Commitment Graph: Data structure tracking active resource leases with expiration times per agent and resource type.

Connected Components: Graph algorithm used in aggressive grouping to find which contentions should be merged. Contentions in same connected component (reachable via paths of overlapping agent sets) form joint contentions.

Contention: Situation where aggregate demand exceeds available supply for one or more resource types. May be single-resource (m=1) or joint (m>1).

ContentionDetector: Component that monitors resource requests, detects when demand exceeds supply, and implements grouping policy to merge overlapping contentions into joints.

Currency: Priority signal in Priority Economy. Earned by releasing resources (quantity × TimeRemaining × DemandMultiplier), spent by burning for increased priority weight.

DemandMultiplier: Scarcity signal computed per resource type as TotalDemand / AvailableSupply. Used in earning formula to incentivize releasing scarce resources.

Embargo Queue: Batching mechanism holding ResourceRequestEvent for fixed window (100ms) before processing, ensuring fairness by eliminating timing advantages.

EmbargoQueue: Component implementing embargo mechanism. Batches requests and publishes ResourceRequestBatchEvent after window expires.

EventBus: Publish-subscribe communication infrastructure enabling loose coupling between components.

Exponential Cone: Mathematical object $K_{exp} = \{(z,y,1) : y>0, y \cdot \exp(z/y) \geq 1\}$ used in Clarabel formulation to represent logarithmic objectives.

Global Pareto Optimality: Property that allocation is Pareto-optimal considering all contested resource types simultaneously (Property 2.3). Achieved through joint arbitration. Contrasts with local Pareto optimality (optimal only within each resource type separately).

Grouping Policy: Strategy for deciding which contentions to merge into joint contentions. Milestone 1 uses aggressive grouping unconditionally. Phase 2 adds configurable policies (k-hop transitive, compatibility matrix, size-bounded).

ImmutableGlobalState: Root of platform state containing resource ledger, agent states, commitment graph, event history, and version number. Immutable once created.

Invariant: Predicate over states that must hold always. Checked by SafetyMonitor before state transitions. Four core invariants: resource conservation, non-negativity, commitment satisfaction, currency accounting.

Joint Contention: Contention spanning multiple resource types and agents, created by grouping policy when single-resource contentions overlap. Arbitrated atomically to achieve global Pareto optimality.

Lease: Temporary resource allocation with expiration time per resource type. Agent must release before expiration or platform reclaims automatically.

Locking Hierarchy: Four-level ordering of locks (state transition > arbitration > agent > resource) that prevents deadlock by preventing circular wait.

Logarithmic Barrier: Mathematical property where $\log(\Phi_i) \to -\infty$ as $\Phi_i \to 0$, preventing starvation by forcing mechanism to allocate to low-utility agents. Foundation of Theorem 3's collusion resistance.

Minting (currency): Creating new currency when resources released. Increases agent's balance. Total currency in system grows through minting, shrinks through burning.

Pareto Optimal: Allocation where no agent can be made better off without making another worse off. Theorem 1 proves Proportional Fairness achieves this. Property 2.3 extends to global optimality across resource types.

Partial Allocation: Allocation between minimum and ideal (min ≤ received ≤ ideal). Requires agent adaptation via adaptToPartialAllocation(). Enables graceful degradation under scarcity.

PreferenceFunction: Agent's utility function $\Phi_i(A)$ expressing how they value allocations. Milestone 1: linear ($\Phi_i = \Sigma_j \, w_{ij} \cdot a_{ij}$). Phase 2: nonlinear (concave, Cobb-Douglas, threshold).

Priority Economy: Closed-loop currency system where agents earn by releasing resources, spend by burning for priority, reaching self-regulating equilibria.

Priority Weight: Weight c_i used in arbitration objective. Formula: c_i = BaseWeight + CurrencyBurned_i. Higher weights receive better allocations across all contested resource types in joint optimizations.

Proportional Fairness: Allocation mechanism maximizing product of utilities (equivalently, sum of logarithms). Provides Pareto optimality with fairness guarantees. Also called Nash Social Welfare.

ResourceBundle: Collection of resources of various types with associated quantities. Used for multi-resource requests, allocations, and releases.

ResourceLedger: Data structure tracking resource ownership per (agent, resource type) pair, currency balances per agent, and lease expirations per (agent, resource type) pair.

ResourceManager: Component enforcing allocations by updating ledger, handling multi-resource releases, calculating earnings, and maintaining resource conservation invariants.

ResourceType: Identifier and properties for a category of resource (compute_units, dataset_access, lab_time_slots, etc.). Platform treats all types uniformly in mechanism.

Rollback: Discarding uncommitted transaction when atomic enforcement fails. Trivial with immutable state—just don't use the working state, retain original state.

Sandbox: Execution environment constraining agent code through timeouts (200ms), thread pool execution (no dedicated threads), and defensive state copying (agents see immutable snapshots).

Semantic Service: Specialized AI capability invoked by agents, treated as a metered resource type by the platform. Examples include semantic paper search, molecular property prediction, and protocol generation. Semantic services have explicit input/output schemas maintaining semantically interpretable domain specialization.

Service: Shorthand for Semantic Service.

Service Composition: Agent workflow where output of one service invocation feeds as input to subsequent service invocations, creating chains (e.g., search → extract → predict → optimize). Platform monitors composition for safety via ServiceCompositionAnalyzer.

ServiceCompositionAnalyzer: Component tracking service invocation patterns, building dependency graphs, measuring composition depth, detecting concerning patterns (excessive depth, dangerous combinations, temporal acceleration), and alerting operators. Provides empirical safety monitoring for service usage.

ServiceInvocationManager: Component managing service invocation lifecycle: accepting requests from agents, debiting service credits, executing backend calls (sync or async), tracking actual costs versus estimates, reconciling credits after completion. Integrates with ResourceLedger and Priority Economy.

ServiceRegistry: Component managing service backend ecosystem: registration and versioning, discovery by category/capability, routing to appropriate backends (local models, external APIs), load balancing, health monitoring. Provides service infrastructure layer between agents and backends.

Service Resource Type: Resource type representing access to specialized AI service, allocated by arbitrator like computational resources. Examples: `semantic_scholar_search_credits`, `molecular_property_prediction_credits`. Participate fully in Priority Economy and joint contention optimization.

Single-Resource Contention: Contention affecting only one resource type (m=1). Special case of general contention model. May be arbitrated independently if agent set is disjoint from all other contentions.

StateTransition: Atomic function mapping (current state, event) → (new state). Creates new immutable state version rather than mutating current.

Structural Sharing: Memory optimization where immutable data structures share unchanged portions between versions, keeping per-transition overhead at O(log(state_size)).

TimeRemaining: Time left on resource lease when agent releases, calculated per resource type. Used in earning formula: earnings_j = quantity_j × TimeRemaining_j × DemandMultiplier_j. Incentivizes early release.

Transaction: Atomic multi-step operation ensuring all-or-nothing semantics. Provided by TransactionManager for joint allocation enforcement (Property 6.3).

TransactionManager: Component providing transaction semantics for atomic operations. Coordinates with SafetyMonitor for pre-commit validation, ensures clean rollback on failure.

Weighted Proportional Fairness: Proportional Fairness with priority weights from currency burning. Objective: maximize $\Sigma_i c_i \cdot \log(\Phi_i(A))$. Enables urgency signaling while maintaining fairness guarantees.

---

14. Conclusion and Implementation Readiness

This architecture document completes the specification for Milestone 1's platform, translating theoretical guarantees into implementable system design. Together with the theory document, these specifications provide everything needed for Milestone 2's implementation phase to begin.

14.1 Architectural Completeness

The architecture specifies the complete system across all dimensions required for implementation:

Component Architecture: Ten major components with clearly defined responsibilities, interfaces, and interaction patterns. Each component is specified at sufficient detail that implementation teams can begin coding immediately.

Data Models: Complete specifications for ImmutableGlobalState, ResourceLedger, AgentConfiguration, Event types, and all supporting data structures. Immutable state semantics are fully explained with structural sharing for efficiency.

Execution Model: Agent lifecycle states, behavioral method contracts, threading model with specific thread allocations and responsibilities, sandbox constraints.

Mechanism Implementation: Proportional Fairness arbitration via Clarabel exponential cones, including joint contention formulation, integer rounding strategy, and error handling for large multi-resource problems.

Priority Economy: Complete earning and spending formulas, DemandMultiplier calculation per resource type, currency balance management with bounded debt, atomic multi-resource releases.

Joint Contention Handling: Aggressive grouping algorithm via connected components analysis, atomic enforcement through TransactionManager, performance characteristics for various problem sizes, Property 6.3 implementation.

Concurrency and Safety: Four-level locking hierarchy with deadlock prevention proof, immutability enabling lock-free reads, safety invariants checked before all transitions, transaction semantics for atomicity.

Configuration and Deployment: Complete YAML schemas for platform and agent configuration, validation rules at three tiers, deployment procedures for single-JVM, monitoring endpoints for operational visibility.

Extension Points: Clearly marked interfaces and abstractions for Phase 2 enhancements (nonlinear preferences, grouping policies, learning modules, Byzantine detection), with conceptual designs showing how extensions build on Milestone 1 foundations.

Validation Support: Architectural support for all six validation scenarios from theory Appendix E, including worst-case joint contention stress test demonstrating scalability limits.

14.2 Key Architectural Properties

The architecture realizes theoretical properties through careful design:

Global Pareto Optimality (Property 2.3) emerges from aggressive grouping of contentions combined with joint optimization. The ContentionDetector's connected components algorithm ensures we never arbitrate contentions separately when they share agents, and the ProportionalFairnessArbitrator's joint formulation optimizes over the full allocation matrix across all contested resource types. No separate code path for "joint optimization"—it's the default formulation, with single-resource as special case m=1.

Atomicity (Property 6.3) is achieved through TransactionManager coordinating with immutable state model. Joint allocation enforcement builds complete new state with all resource transfers, currency burns, and lease settings applied together, then commits atomically or rolls back cleanly. Immutability makes rollback trivial—just don't use the working state if validation fails.

Computational Tractability (Theorem 2) is preserved through exponential cone formulation regardless of joint problem size. The arbitrator doesn't have special-case logic for large joints—it formulates the problem with appropriate dimensions and calls Clarabel, which provides polynomial-time guarantees. For Milestone 1's scale ($n \leq 20$, $m \leq 10$, up to 200 variables), solve time remains <500ms at 99th percentile.

Collusion Resistance (Theorem 3's heuristic argument) is structural—the logarithmic objective function makes starvation attempts self-defeating regardless of joint contention size. When coalitions try to monopolize multiple resource types simultaneously through joint optimization, the mechanism responds by allocating to victims across all contested types, undermining the coalition's goals.

Deadlock Freedom follows from the locking hierarchy enforcing acyclic wait-for graphs. Joint allocation enforcement might acquire many locks (multiple agent locks, multiple resource locks), but the sorted acquisition order within each level prevents cycles.

Starvation Protection through BaseWeight=10.0 ensures even agents with zero currency participate meaningfully. In joint contentions, this protection applies across all contested resource types—low-currency agents receive modest allocations for all types, not excluded from any.

These properties are not emergent from careful tuning but rather structural consequences of architectural decisions. The architecture was designed specifically to realize the theoretical guarantees.

14.3 Implementation Priorities for Milestone 2

Milestone 2 teams should implement components in dependency order, validating each before building the next:

Phase 1.2.1: Foundation
- ImmutableGlobalState and data models (ResourceLedger, AgentConfiguration, Event types)
- EventBus and basic publish-subscribe infrastructure
- PlatformRuntime skeleton (initialization, basic event loop, shutdown)
- Safety invariants and SafetyMonitor

Phase 1.2.2: Core Mechanism
- ContentionDetector with aggressive grouping via connected components
- PriorityWeightCalculator with multi-resource earning formula
- ProportionalFairnessArbitrator with Clarabel integration for joint problems
- TransactionManager for atomic enforcement
- ResourceManager with atomic joint allocation enforcement

Phase 1.2.3: Agent Support
- AgentExecutionManager with timeout enforcement
- Agent configuration loading and validation
- EmbargoQueue for fair request batching
- Example agent implementations (3-5 agents with diverse preferences)

Phase 1.2.4: Validation
- Implement all six validation scenarios from theory Appendix E
- Run scenarios, collect metrics, generate reports
- Compare empirical results to theoretical predictions
- Identify discrepancies for investigation

This staged approach enables early validation: after Phase 1.2.1, we can test state transitions and safety properties. After Phase 1.2.2, we can run basic arbitrations. After Phase 1.2.3, we have a complete system ready for comprehensive validation.

14.4 Next Steps

Milestone 1 informs Milestone 2's continuation on the project:

Implementation (Milestone 2):
- Platform v0.1 implemented in Java, running on single JVM
- All ten core components implemented with specified interfaces
- Joint contention handling with aggressive grouping operational
- Atomic multi-resource allocation enforcement validated

- Priority Economy earning and burning functional
- 3-5 example agents demonstrating diverse preferences across resource types

Validation (Milestone 2):
- All six validation scenarios executable and passing
- Theoretical properties validated empirically:
  - Pareto improvements achieved in >90% of arbitrations
  - Individual rationality holds (all agents ≥ 120% of independent baseline)
  - Semantic divergence within bounds (95% of agents within $2 \cdot ||\Phi_{hidden}||_\infty$)
  - Collusion attempts fail (coalition ≤100% of honest baseline)
  - Worst-case joint contention completes <500ms at 95th percentile
- Performance targets met:
  - Arbitration latency <200ms at 95th percentile for typical contentions
  - <500ms for worst-case joint contentions
  - Throughput >50 arbitrations/second sustained
  - Lock contention <20% of execution time

Together, theory + architecture constitute Milestone 1 complete, and + implementation + validation constitute Phase 1 complete.

14.5 From Architecture to Code

The path from this specification to working code is direct:

For each component: The interface defines method signatures, the prose explains responsibilities and design rationale, the code examples show key algorithms. Implementation teams translate these specifications into Java classes, using the specified interfaces and following the documented patterns.

For data models: The structure diagrams show fields and relationships, the immutability requirements dictate use of persistent data structures (PCollections or Vavr libraries), the validation rules define what's legal. Implementation creates corresponding Java records or classes with specified structure and validation.

For algorithms: The pseudocode (contention grouping via connected components, joint problem formulation for Clarabel, atomic enforcement pattern, rounding and feasibility restoration) provides step-by-step logic. Implementation translates to Java, handling edge cases and errors as specified.

For configuration: The YAML schemas define structure, the validation rules define semantics, the examples show valid configurations. Implementation uses standard YAML parsing libraries (SnakeYAML or Jackson YAML), validates against schemas, and constructs configuration objects.

The architecture intentionally avoids implementation details that would constrain choices unnecessarily (exact library versions, specific optimization tricks, detailed exception handling for every method) while providing enough detail that implementations will interoperate (interfaces must match, event schemas must align, validation rules must agree). This balance enables

multiple teams to implement concurrently (different components by different developers) with confidence that their implementations will integrate successfully.

Quality Standards:

Implementations should prioritize correctness over cleverness. The architecture is designed for clarity and verifiability, not for maximum performance or minimal code size. Implementations should:

- Follow the specified interfaces exactly (method signatures, exception contracts, return types)
- Implement the specified algorithms clearly (readable code with comments, even if slightly verbose)
- Validate inputs and check preconditions thoroughly (defensive programming for safety-critical platform)
- Log extensively at appropriate levels (DEBUG for detailed traces, INFO for major events, WARN for unexpected conditions, ERROR for failures)
- Test comprehensively (unit tests per component, integration tests for flows, property tests for invariants)

The goal is code that obviously works, not code that might work if you squint. For a coordination platform that might eventually coordinate powerful AI systems, correctness is paramount.

14.6 Looking Forward: From Phase 1 to Phase 2 and Beyond

This Phase 1 architecture is deliberately constrained to establish clean foundations with provable properties. The constraints—linear preferences, discrete resources, single-JVM, non-adversarial agents, unconditional aggressive grouping—enable exact optimization with strong guarantees but limit expressiveness and scale.

Phase 2 relaxes these constraints systematically:

Expressiveness extensions (nonlinear preferences, continuous resources) add richness while carefully preserving convexity where possible, using approximation with bounds where necessary.

Performance extensions (grouping policies, parallel arbitration, approximate methods) enable scaling beyond n=20 by trading some optimality for computational efficiency, with formal approximation bounds showing exactly what's sacrificed.

Intelligence extensions (learning modules, predictive arbitration) transform the platform from reactive to proactive, anticipating contentions before they occur and positioning resources optimally.

Security extensions (classloader isolation, Byzantine detection, formal coalition-proofness) harden the platform against adversarial agents, moving from "assume honesty" to "verify trust."

Each phase is essentially a rebuild incorporating learnings from the previous phase. This might seem inefficient, but for a safety-critical platform in a rapidly evolving field, the flexibility to make

fundamental changes based on empirical learnings is more valuable than code reuse. With modern development tools and LLM-assisted coding, rebuilding is cheap; discovering fundamental design flaws late is expensive.

The architecture document you've read provides the blueprint for Phase 1. Milestone 2 implements this blueprint in working Java code. Milestone 3 refines based on empirical learnings. And each step brings us closer to coordination mechanisms ready for the powerful AI systems that might help us coordinate safely in the near future.