Multi-Agent Harnessing Architecture and Orchestration Patterns in Specialized Autonomous Systems

With Applications to Agentic Coding for Industry-Grade SaaS, B2B, and D2C Products

Sagnik Mitra alias sgnk

May 2026

Abstract

Multi-agent systems are becoming the default architecture for complex autonomous software, robotics, cyber-defense, financial operations, and AI-native product engineering. Yet most implementations fail because they multiply agents without designing the harness that controls authority, state, verification, memory, observability, and failure recovery. This paper presents a practical architecture for harnessed multi-agent autonomy: control planes, task graphs, agent contracts, orchestration patterns, runtime guardrails, verification loops, and product-grade workflows. It also maps these concepts to agentic coding teams that can build reliable SaaS, B2B, and D2C products using specialized agents for product, design, architecture, backend, frontend, security, QA, DevOps, observability, and release governance. The core claim is simple: agents may be autonomous inside their lane, but the system must never be uncontrolled as a whole.

Keywords: multi-agent systems, agent orchestration, autonomous systems, agent harness, agentic coding, SaaS architecture, B2B products, D2C products, observability, verification, AI governance.

1 The Problem: More Agents Are Not Architecture

The current industry mistake is predictable: teams add a planner agent, a researcher agent, a coder agent, a reviewer agent, and an executor agent, then assume they have a multi-agent system. They do not. They have distributed prompt choreography.

A specialized autonomous system requires a stronger discipline. It must define who has authority, who owns state, who may use which tool, who verifies outputs, when humans enter the loop, and what happens when the system becomes uncertain. NIST describes AI risk management as a lifecycle problem involving the design, development, use, and evaluation of AI systems, not merely a model selection problem [1]. That framing matters: agentic systems act over time, across tools, and across changing state.

> **Hard truth:** the harness matters more than the agents. Agents are replaceable cognitive workers. The harness is the operating system that makes them safe, inspectable, and commercially deployable.

2 Core Definitions

Agent. A bounded actor that can observe context, reason over a task, produce artifacts, and optionally invoke tools.

Harness. The runtime control envelope around agents: policy, permissions, budgets, schemas, memory, verification, observability, and recovery.

Orchestration. The mechanism that decomposes goals, routes work, schedules execution, merges artifacts, and coordinates state transitions.

Specialized autonomous system. A system where multiple bounded agents perform domain-specific work under explicit governance to achieve a mission with limited human intervention.

Agentic coding. The use of AI agents as structured software delivery workers: product analyst, solution architect, designer, developer, tester, security reviewer, release manager, and site reliability observer.

3 Reference Architecture

A production-grade multi-agent system should be read as a controlled runtime, not a chatroom. The minimum viable architecture contains mission intake, governance, orchestration, specialist agents, state, tools, verification, and observability.

The reference architecture in Figure 1 deliberately separates cognition from authority. The system may use probabilistic agents, but its commits, permissions, traces, and risk gates should be deterministic enough to audit.

4 The Five Architectural Planes

Serious agent systems need five planes. Weak systems mix all of them inside prompts.

4.1 Control Plane

The control plane owns authority. It decides delegation, cancellation, escalation, approval, and recovery.

4.2 Data Plane

The data plane moves events, messages, retrieved documents, tool outputs, telemetry, and artifacts between services.

4.3 State Plane

The state plane owns truth: task status, locks, memory snapshots, customer records, deployment state, runbooks, and audit trails.

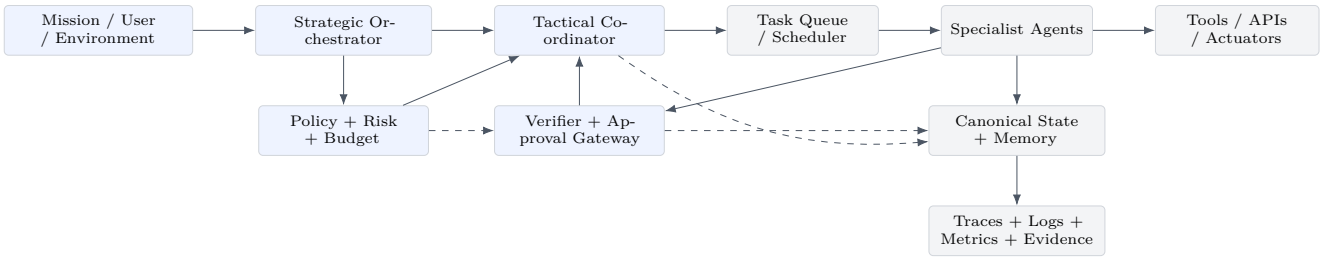

Figure 1: Reference architecture for a harnessed specialized autonomous system. The control loop is explicit: mission intake, policy, orchestration, bounded execution, independent verification, state commit, and observability.

4.4 Execution Plane

The execution plane performs work: LLM calls, code edits, browser automation, database queries, API calls, shell execution, and robotic motion.

4.5 Assurance Plane

The assurance plane constrains risk: policy, verification, adversarial testing, sandboxing, observability, rollback, and human approval. OWASP explicitly recommends least privilege, validated output formats, human approval for high-risk operations, and adversarial testing for prompt injection risk [4].

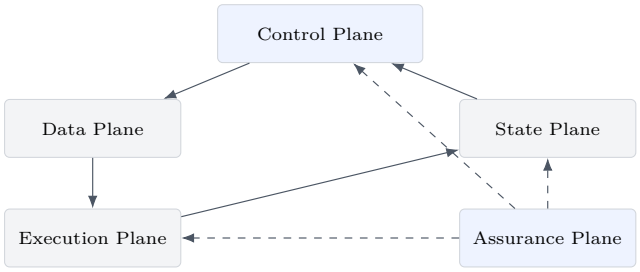

Figure 2: The five planes of agentic autonomy.

5 Agent Role Contracts

Agents should not be defined by motivational personality prompts. They should be defined by contracts.

Listing 1: Example agent contract

```
agent:
  id: verifier.payment_policy.v1
  role: independent_verifier
  allowed_inputs:
    - proposed_action
    - policy_context
    - state_snapshot
  allowed_tools:
    - policy_lookup
    - transaction_limit_checker
  forbidden_tools:
    - payment_execute
    - database_write
  output_schema:
    decision: approve | reject | repair | escalate
    reasons: string[]
    risk_score: number
    human_approval_required: boolean
```

The contract should declare allowed tools, forbidden tools, input schemas, output schemas, budget limits, retry limits, verification rules, and escalation thresholds. The narrower the agent boundary, the easier the system is to debug.

6 Core Orchestration Patterns

No single orchestration pattern solves every domain. Mature systems combine patterns based on risk, latency, auditability, and resource constraints.

6.1 Hierarchical Supervisor Pattern

The hierarchical pattern uses a supervisor to decompose the mission and assign bounded tasks to specialists. It is the safest default for SaaS, finance, legal-tech, healthcare, cybersecurity, and B2B workflow automation.

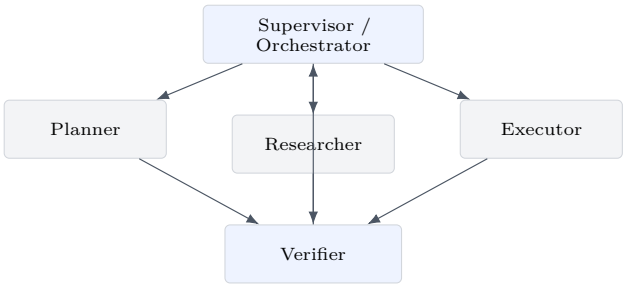

Figure 3: Hierarchical supervisor pattern.

Strengths: clear accountability, strong auditability, simple failure containment. **Weaknesses:** central bottleneck and reduced adaptability.

6.2 Blackboard Pattern

In the blackboard pattern, agents contribute to a shared structured workspace. This is useful for diagnostics, research synthesis, threat analysis, scientific reasoning, product discovery, and UX decision-making.

Listing 2: Blackboard entry schema

```
blackboard_entry:
  id: string
  author_agent: string
  claim: string
  evidence_refs: string[]
  confidence: number
  expiry: timestamp
  state_version: string
  verification_status: unverified | verified | rejected
```

Without schemas, blackboards rot into noisy group chats. Every claim must have evidence, author, confidence, expiry, and verification status.

6.3 Event-Driven Mesh Pattern

Event-driven systems use an event bus to decouple producers and consumers. They scale well, but causal reasoning

becomes difficult unless traces are mandatory. OpenTelemetry frames observability around signals such as traces, metrics, logs, baggage, and profiles [2]; those concepts map directly to multi-agent runtime instrumentation.

Listing 3: Agent event envelope

```
{
  "event_id": "evt_123",
  "event_type": "task.completed",
  "mission_id": "mission_456",
  "trace_id": "trace_789",
  "source_agent": "execution_agent.v1",
  "state_version": "state_88",
  "risk_level": "medium",
  "payload": {}
}
```

Event-driven agent systems require idempotency, schema versioning, dead-letter queues, causal tracing, retry budgets, and loop prevention.

6.4 Market or Auction-Based Allocation

Auction-based allocation is useful when agents have different capabilities, cost, latency, risk, availability, or location. This is common in robot fleets and distributed work scheduling. Recent robot fleet work emphasizes shared world state, planning, scheduling, execution, and replanning across heterogeneous robots [6].

$$Score = 0.30C + 0.20Q - 0.20K - 0.15L - 0.15R \quad (1)$$

Where C is capability match, Q is confidence, K is normalized cost, L is latency, and R is risk. Do not select agents only on speed or cost. That creates unsafe shortcuts.

6.5 Hybrid Pattern

Serious systems usually end up hybrid: hierarchical control for governance, auction allocation for dynamic work, blackboard memory for shared reasoning, and event streams for tool results and telemetry.

Function	Best Pattern
Mission control	Hierarchical supervisor
Dynamic allocation	Auction / market
Shared reasoning	Blackboard
Tool propagation	Event-driven mesh
Safety approval	Verifier + human gate
Recovery	Supervisor + event replay
Memory	Canonical state + event log

Table 1: Pattern selection by system function.

7 Mission-to-Commit Workflow

The central workflow is not "agent talks to agent." It is mission intake, policy check, task graph creation, execution, verification, and state commit.

8 Harness Layer Deep Dive

The harness is the runtime control envelope. It makes probabilistic agents behave inside deterministic boundaries.

A harness should include: identity and access control, tool whitelists, data egress controls, prompt and content isolation, output validation, risk scoring, budget ceilings, approval gates, sandboxed execution, rollback, and immutable audit logs. LangGraph's public documentation emphasizes durable execution, persistence, human-in-the-loop control, and runtime visibility as core orchestration capabilities [3].

9 Verification Workflow

The executor must not verify itself. Verification should be independent and risk-weighted.

Action	Required Verification
Read-only retrieval	Provenance + schema
Internal planning	Logical consistency
Database write	Schema + permission + state version
Code execution	Sandbox + tests + static analysis
Payment or trade	Policy + risk + human approval
Customer message	Factuality + tone + compliance
Production deploy	CI + security + rollback plan

Table 2: Risk-weighted verification matrix.

Verification includes schema verification, policy verification, factual verification, logical verification, tool verification, security verification, and human verification.

10 State and Memory Architecture

Multi-agent systems fail when state becomes vague. Use three layers: local working memory, shared mission workspace, and canonical state.

Agents should not directly mutate canonical state. They submit proposed state transitions.

Listing 4: State transition proposal

```
state_transition:
  transition_id: string
  proposed_by: agent_id
  previous_state_version: string
  operation: create | update | delete | reserve | release
  target_resource: string
  payload: object
  verification_status: pending
```

11 Observability Requirements

Once agents act asynchronously, explainability becomes a distributed-systems problem. Every mission, task, tool call, agent decision, verification decision, human approval, and state transition needs a trace ID.

Listing 5: Minimum trace record

```
{
  "trace_id": "trace_abc",
  "mission_id": "mission_001",
  "task_id": "task_009",
  "agent_id": "planner.v2",
  "state_version": "state_104",
  "tool_name": "database.write",
  "policy_bundle": "policy.finance.v3",
  "risk_score": 0.72,
  "decision": "request_human_approval",
  "latency_ms": 1200,
  "tokens_used": 3400,
  "retry_count": 1
}
```

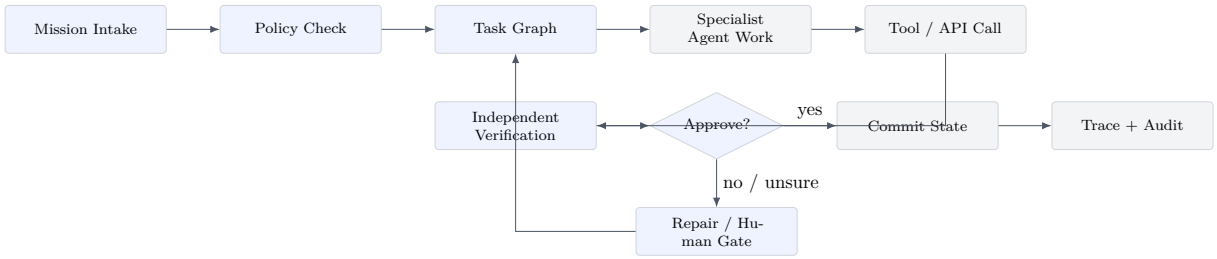

Figure 4: Mission-to-commit workflow. State is committed only after explicit verification.

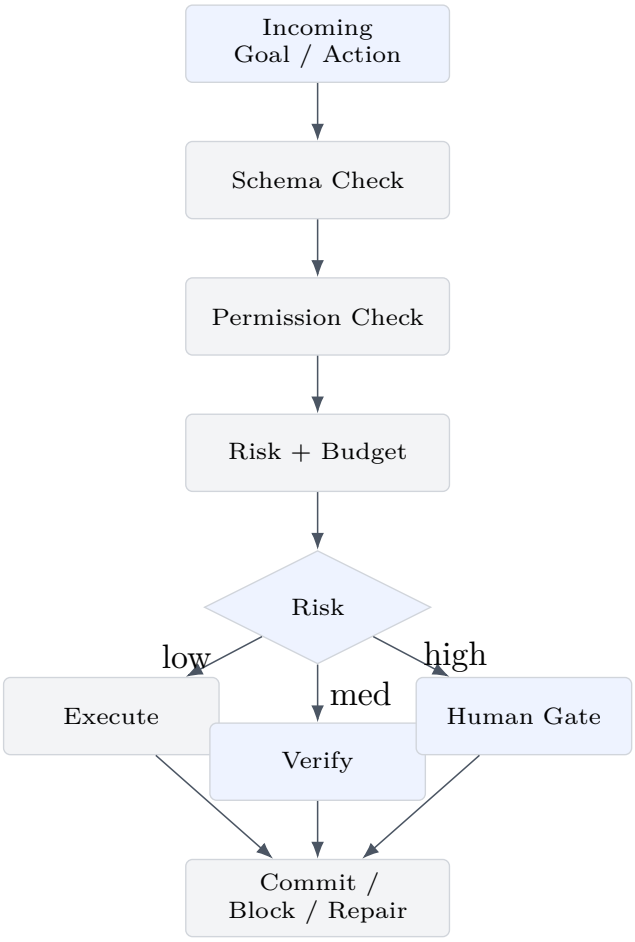

Figure 5: Runtime harness gate.

Track agent failure rate, verifier rejection rate, retry count, human escalation rate, state conflict rate, dead-letter events, tool error correlation, budget burn, and mean time to recovery.

12 Failure and Recovery Workflows

A serious system must design the unhappy path before celebrating autonomy.

13 Human-in-the-Loop Design

Human approval is not a vague "Should I continue?" prompt. It is a structured decision packet.

Review packet minimum: mission ID, task ID, proposed action, expected impact, risk score, passed/failed policy checks, evidence references, alternatives, recommendation, and rollback plan.

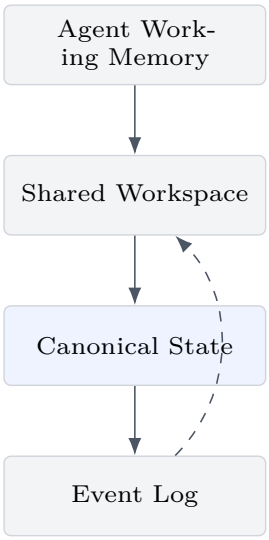

Figure 6: Memory and state layers.

Failure	Recovery
Hallucinated output	Verifier rejects; repair or reassign
Tool timeout	Retry with backoff; fallback tool
State conflict	Lock reconcile; reload state
Infinite loop	Retry budget kill switch
Prompt injection	Isolate content; block tool call
Unsafe action	Policy stop + human review
Compromised agent	Revoke identity and tools

Table 3: Common failure modes and recovery paths.

Humans should approve actions with known blast radius, evidence, rollback plan, and alternatives. Anything less is fake consent.

14 Agentic Coding for Industry-Grade Products

Agentic coding is where these concepts become commercially useful. But the goal is not faster code generation. The goal is faster product delivery without collapsing quality, security, maintainability, or user trust.

A real SaaS, B2B, or D2C product requires far more than screens and endpoints. It needs authentication, authorization, tenant isolation, billing, onboarding, analytics, observability, support workflows, data privacy, audit logs, CI/CD, rollback, cost control, and a product feedback loop.

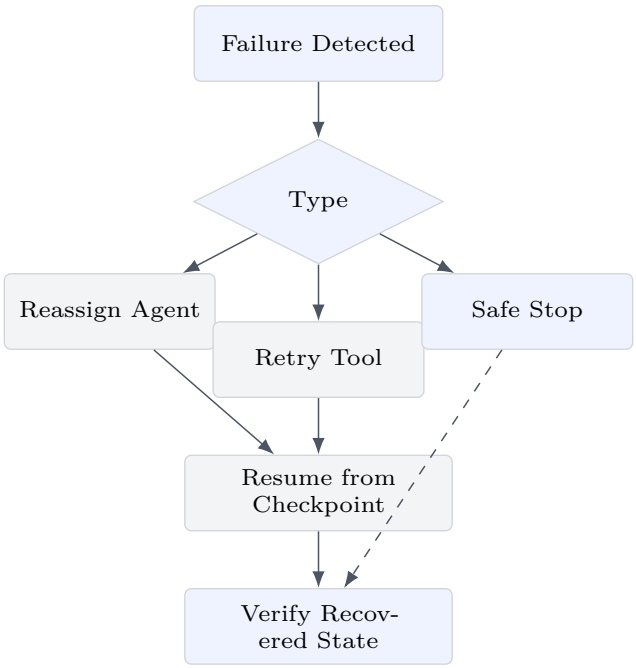

Figure 7: Failure recovery workflow.

14.1 Agentic Product Factory

A strong agentic coding setup turns the software team into a harnessed product factory.

14.2 Agent Roles in Product Engineering

Agent	Responsibility
Product Agent	PRD, use cases, acceptance criteria, prioritization
Research Agent	Competitor review, user pain, market evidence
UX Agent	Flows, IA, wireframes, design system mapping
Architecture Agent	System design, data model, service boundaries
Backend Agent	APIs, database, jobs, integrations, domain logic
Frontend Agent	UI, state management, accessibility, responsiveness
QA Agent	Test plan, unit/integration/e2e tests, edge cases
Security Agent	Threat model, auth, RBAC, secret handling, dependency risk
DevOps Agent	CI/CD, environment config, preview deploys, rollback
Verifier Agent	Cross-checks requirements, diff, tests, risk, release gates

Table 4: Specialized coding agents mapped to product delivery.

14.3 B2B SaaS Pattern

B2B SaaS is governance-heavy. The agentic system must prioritize tenant isolation, RBAC, auditability, data export, admin controls, integration reliability, and contractual workflows.

Workflow:

1. Product Agent creates an account hierarchy: organization, workspace, role, permission, user, invitation.
2. Architecture Agent defines tenant boundaries and data access policy.
3. Backend Agent builds typed APIs and migration-safe schemas.
4. Security Agent verifies RBAC and cross-tenant leakage paths.
5. QA Agent builds permission matrix tests.
6. DevOps Agent adds preview deploys and rollback.
7. Verifier Agent blocks release if tenant isolation or audit logging is missing.

14.4 D2C Product Pattern

D2C products are conversion-heavy. The agentic system must prioritize onboarding, activation, payments, retention, recommendations, funnel analytics, performance, mobile UX, and trust signals.

Workflow:

1. Product Agent defines acquisition, activation, retention, referral, and revenue metrics.
2. UX Agent designs onboarding, empty states, checkout, and support journeys.
3. Frontend Agent implements responsive UI and speed budgets.
4. Backend Agent builds user profile, order, subscription, and notification services.
5. QA Agent tests conversion-critical flows.
6. Observability Agent instruments funnels and drop-off points.
7. Growth Agent proposes experiments, but Verifier Agent prevents dark patterns.

14.5 Industry-Grade Gates

Agentic coding must use quality gates. Otherwise it only produces faster technical debt.

Gate	Release Standard
Requirement gate	PRD, edge cases, measurable acceptance criteria
Architecture gate	Data model, API contract, scalability assumptions
Security gate	Auth, RBAC, secrets, dependency scan, threat model
Quality gate	Tests, lint, type checks, performance budgets
UX gate	Accessibility, responsive states, error states
Ops gate	Logs, traces, metrics, alerts, rollback
Business gate	Pricing, onboarding, support, analytics, retention

Table 5: Release gates for agentic product engineering.

14.6 Agentic Coding Control Loop

14.7 What Agentic Coding Should Never Do

An agentic coding system should not let the same agent define requirements, write code, approve the diff, disable tests, deploy to production, and interpret production failures. That is not autonomy. That is concentrated unchecked risk.

The secure model is separation of duties:

- Builder agents create artifacts.
- Verifier agents challenge artifacts.

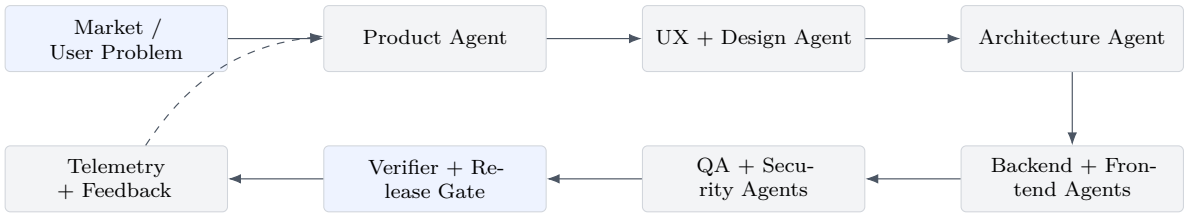

Figure 8: Agentic product factory for SaaS, B2B, and D2C delivery. The loop does not end at code. It ends at telemetry-backed product learning.

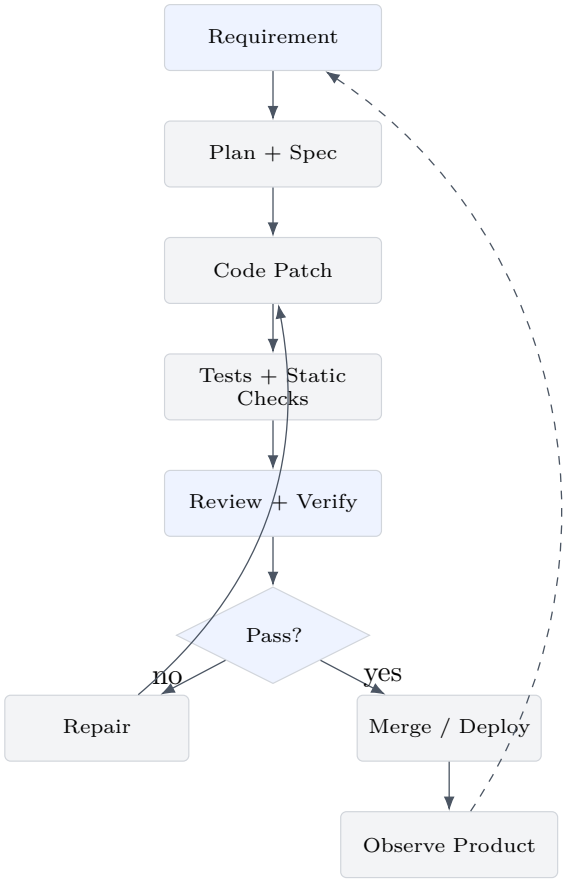

Figure 9: Agentic coding loop. The loop includes product telemetry, not just code completion.

- Policy services define allowed actions.
- CI systems enforce deterministic checks.
- Humans approve high-risk production transitions.

MCP can help standardize how coding agents connect to external systems such as files, databases, tools, and workflows [5], but standard connectivity increases the importance of least privilege and tool isolation.

15 Implementation Stack

A practical implementation can use the following stack.

Layer	Typical Choices
Orchestration	LangGraph, Temporal, custom DAG runtime
Agent workers	Containerized Python/Node workers
State	Postgres, Redis, event store
Memory	pgvector, Qdrant, object store
Messaging	Kafka, NATS, RabbitMQ, Redis streams
Policy	OPA, Cedar, custom policy service
Observability	OpenTelemetry, Prometheus, Grafana, Jaeger
Execution sandbox	Docker, Firecracker, gVisor
Product repo	GitHub, CI, protected branches, preview deployments
Human gate	GitHub PR, Slack approval, internal console

Table 6: Practical stack for harnessed agentic systems.

16 Autonomy Maturity Model

Most teams claim L4 while operating at L2 with extra prompts. The difference is not the number of agents; it is governance, verification, state, observability, and recovery.

Level	Capability
L0	Human uses tools manually
L1	Single assistant suggests actions
L2	Agent uses tools under human control
L3	Multiple agents coordinate tasks
L4	Harnessed autonomy with verification and state
L5	Adaptive governed autonomy with self-repair

Table 7: Agentic autonomy maturity levels.

17 Design Rules

1. Start hierarchical before adding swarm behavior.
2. Define canonical state before building agent memory.
3. Define agent contracts before writing prompts.
4. Add verification before adding more agents.
5. Separate planning, execution, and approval.
6. Use traces for every agent and tool call.
7. Treat external content as untrusted.
8. Sandbox code execution and browser automation.
9. Use human gates for high-impact actions.
10. Optimize for recoverability, not demo magic.

18 Conclusion

Multi-agent systems are not powerful because they contain many agents. They are powerful when responsibilities are partitioned, autonomy is bounded, authority is explicit, state is coherent, tools are constrained, verification is independent, failures are recoverable, and observability is complete.

For agentic coding, the same principle applies. The goal is not to make one AI write an entire product in a burst. The goal is to create a controlled product engineering system where specialized agents collaborate under contracts and gates to ship reliable SaaS, B2B, and D2C products.

Final principle: agents should be autonomous inside their lane, but the system must never be uncontrolled as a whole. That is the difference between an industry-grade autonomous system and a distributed hallucination machine.

References

[1] National Institute of Standards and Technology. *AI Risk Management Framework*. NIST states that AI RMF is intended for voluntary use and for incorporating trustworthiness into the design, development, use, and evaluation of AI products, services, and systems. https://www.nist.gov/itl/ai-risk-management-framework

[2] OpenTelemetry. *Signals*. Official documentation describing observability signals including traces, metrics, logs, baggage, and profiles. https://opentelemetry.io/docs/concepts/signals/

[3] LangChain. *LangGraph Overview*. Documentation describing LangGraph as an orchestration runtime for long-running, stateful agents with durable execution, streaming, human-in-the-loop, and persistence. https://docs.langchain.com/oss/python/langgraph/overview

[4] OWASP GenAI Security Project. *LLM01:2025 Prompt Injection*. Official guidance on direct and indirect prompt injection, prevention, least privilege, output validation, human approval, and adversarial testing. https://genai.owasp.org/llmrisk/llm01-prompt-injection/

[5] Model Context Protocol. *What is the Model Context Protocol?*. Official documentation defining MCP as an open-source standard for connecting AI applications to external systems, data sources, tools, and workflows. https://modelcontextprotocol.io/docs/getting-started/intro

[6] Gupta, R., Asbery, T., Merchant, Z., Anwar, A., and Thomason, J. *RobotFleet: An Open-Source Framework for Centralized Multi-Robot Task Planning*. arXiv, 2025. https://arxiv.org/abs/2510.10379

[7] Datta, S., Nahin, S. K., Chhabra, A., and Mohapatra, P. *Agentic AI Security: Threats, Defenses, Evaluation, and Open Challenges*. arXiv, 2025. https://arxiv.org/abs/2510.23883