Modeling Application Security Processes with *Work the System*, *Traction*, and *This Is Service Design Doing*

A Comprehensive Guide to Building Mature, Developer-Centric Security Programs

Jordan Suber

Updated: November 30, 2025

Abstract

This document provides a comprehensive framework for modeling Application Security (AppSec) processes by synthesizing insights from three influential books: *Work the System* (Sam Carpenter), *Traction* (Gino Wickman), and *This Is Service Design Doing*. It offers practical guidance for transforming ad-hoc security tooling into coherent, documented systems that integrate seamlessly into modern DevSecOps workflows. The guide includes service blueprints, standard operating procedures (SOPs), metrics frameworks, maturity model alignment, and templates for immediate implementation.

Document Navigation

Contents

1 Purpose of This Document

1.1 Goals

This document serves three primary objectives:

1. **Synthesize management frameworks for AppSec**: Demonstrate how three books—*Work the System*, *Traction*, and *This Is Service Design Doing*—provide complementary perspectives for building mature security programs.
2. **Provide actionable artifacts**: Deliver service blueprints, SOPs, metrics frameworks, and templates that can be adapted immediately to real-world AppSec programs.
3. **Bridge tooling and process**: Move organizations from tool-centric thinking ("We use GHAS and Snyk") to systems thinking ("We operate a coherent AppSec program with defined processes, ownership, and continuous improvement").

1.2 Target Audience

This guide is designed for:

- AppSec engineers and architects seeking to formalize existing practices,
- Security managers building or scaling security programs,
- DevOps/Platform engineers integrating security into CI/CD pipelines,
- Engineering leadership seeking to understand and invest in AppSec maturity.

1.3 The Transformation

The intent is to provide an actionable path from:

> "We have security tools and gates"
> *to*
> "We operate a coherent, documented AppSec **system** that delivers measurable security outcomes with minimal developer friction."

Key Insight

The most effective AppSec programs treat security as an **internal service** to development teams—with clear interfaces, documented processes, measurable SLAs, and continuous improvement loops.

1.4 Document Scope

This document covers:

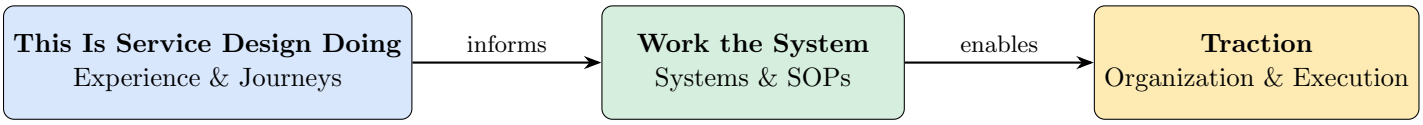
- Secure software development lifecycle (SSDLC) processes,
- CI/CD security integration (SAST, SCA, secrets scanning, container security),
- Vulnerability management and triage,
- Supply chain security,
- Security incident response for application-layer findings,
- Developer enablement and security champion programs.

Out of scope: network security, infrastructure security operations, identity and access management (IAM), and compliance audit programs (though these may interface with AppSec processes).

Navigation tip. If you already know these books, skim Section 2 and jump to Section 4 for the modern tooling context, then proceed to the blueprints and SOPs in Sections 5–6.

2 How Each Book Helps with AppSec Process Modeling

The three books provide complementary lenses:



Together: A complete model for AppSec program design

2.1 *This Is Service Design Doing*

Core Idea

This Is Service Design Doing is a practical guide to service design that emphasizes:

- Understanding users and stakeholders through research,
- Mapping **customer journeys** to reveal pain points and opportunities,
- Designing services using **service blueprints**,
- Prototyping and iterating based on feedback.

A **service blueprint** typically includes five key swim lanes:

1. **Customer actions**: What the user (developer) does,
2. **Frontstage actions**: What the user sees and interacts with,
3. **Backstage actions**: What the organization does behind the scenes,
4. **Supporting processes**: Systems and integrations that enable the service,
5. **Physical/Digital evidence**: Artifacts, communications, logs, and dashboards.

Application to AppSec

This book is most valuable when your goal is:

> **Design AppSec as a developer-centered service**—with clear journeys, well-defined touchpoints, responsive support, and measurable outcomes.

Key applications include:

- **Developer journey mapping**: Understanding how developers experience security controls throughout the SDLC—from IDE to production.
- **Friction identification**: Pinpointing where security processes slow down delivery or create confusion.
- **Touchpoint optimization**: Improving how security findings are communicated (PR annotations, dashboards, notifications).
- **Support channel design**: Creating clear escalation paths and self-service resources.

Action Item

Create a developer journey map for your most common workflow (e.g., feature development with PR). Identify every point where developers interact with security tooling or policies.

2.2 *Work the System*

Core Idea

Work the System argues that every organization is a collection of **systems** and **subsystems**. To improve results consistently, you must:

- Step outside the "whirlwind" of daily operations to see the system,
- Identify and isolate key systems,
- Document them as simple, written procedures,
- Improve those procedures incrementally based on real-world feedback.

The book emphasizes three foundational documents:

1. **Strategic Objective**: A concise statement of what success looks like.
2. **Operating Principles**: Decision-making guidelines that reflect organizational values.
3. **Working Procedures**: Step-by-step documentation of how recurring work is performed.

Application to AppSec

Instead of viewing AppSec as a collection of tools (GHAS, Snyk, Semgrep, etc.), *Work the System* encourages you to define and document the **systems** that orchestrate those tools:

- **Inputs**: Events, triggers, and artifacts that initiate the process (e.g., PR opened, dependency updated, vulnerability published).
- **Process**: Steps, decisions, decision trees, and responsible parties.
- **Outputs**: Approvals, tickets, metrics, notifications, and audit records.

For AppSec, this translates to:

- **Strategic Objective example**: "We detect and remediate critical application vulnerabilities before they become exploitable in production, with minimal friction for developers and measurable improvement quarter over quarter."
- **Operating Principles examples**:
  - "Security checks integrate into existing developer workflows—we shift left, not block."
  - "We prioritize remediation by exploitability and business impact, not just severity scores."
  - "Processes are documented in plain language and kept as short as practical."
  - "Exceptions require documented justification and have expiration dates."
- **Working Procedures**: SOPs for Secure PR Flow, Vulnerability Triage, Secrets Incident Response, Dependency Management, Exception Handling, and more.

Key Insight

The power of *Work the System* is forcing explicit documentation of **who does what, when, and how**—turning tribal knowledge into repeatable, improvable systems.

2.3 *Traction*

Core Idea

Traction introduces the Entrepreneurial Operating System (EOS), which structures organizational execution around six components:

1. **Vision**: Shared understanding of organizational direction.
2. **People**: Right people in the right seats with clear accountability.
3. **Data**: Simple, objective metrics that drive decisions.
4. **Issues**: Systematic identification and resolution of problems.
5. **Process**: Documented and consistently followed systems.
6. **Traction**: Disciplined execution through cadence and accountability.

Key EOS tools include:

- **Accountability Chart**: Who owns what (not just an org chart—role-based ownership).
- **Rocks**: 90-day priorities that move the needle.
- **Scorecard**: Weekly metrics that indicate process health.
- **Level 10 (L10) Meetings**: Structured weekly meetings for issue resolution.
- **IDS (Identify, Discuss, Solve)**: Problem-solving framework.

Application to AppSec

Traction provides the organizational scaffolding to ensure AppSec is not "just a set of tools" but a managed function with:

- **Clear ownership**:
  - Who owns the secure SDLC?
  - Who owns CI/CD security gates?
  - Who owns vulnerability management?
  - Who owns supply chain security?
- **Defined Rocks (90-day priorities)**:
  - "Achieve 100% SAST coverage across production repositories."
  - "Reduce critical vulnerability MTTR from 30 days to 14 days."
  - "Launch security champion program with 10 trained champions."
  - "Implement SBOM generation for all container images."
- **Scorecard metrics**:
  - Open critical/high vulnerabilities by age bucket,
  - Mean time to remediate (MTTR) by severity,
  - Percentage of repos with security scanning enabled,
  - Number of secrets detected in code (trend),
  - Developer friction score (survey-based).
- **Leadership cadence integration**:
  - AppSec metrics appear on executive scorecards,
  - AppSec issues are raised in L10 meetings,
  - AppSec Rocks are reviewed quarterly at leadership level.

Action Item

Create an AppSec accountability chart mapping every key process to an explicit owner. Ensure no process is orphaned.

3 Recommended Reading Order for AppSec Modeling

If your specific goal is to model and mature Application Security processes, a practical reading order is:

1. ***This Is Service Design Doing*** (Stickdorn et al.)

   Start here to learn tools for mapping journeys and designing services. Your first deliverable should be one or two **service blueprints** for key AppSec flows (e.g., Secure PR, Vulnerability Triage). These blueprints provide a developer-centered view of how security appears in daily work.

   Focus chapters: Journey Mapping, Service Blueprinting, Prototyping.

2. ***Work the System*** (Sam Carpenter)

   Use this to transform blueprints into **SOPs**—clear, written procedures that can be followed and improved. You move from "this is the developer journey" to "this is the documented system we operate."

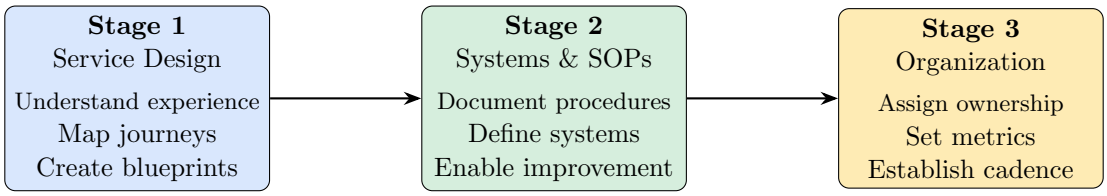
   Focus chapters: The Systems Mindset, Strategic Objective, Working Procedures.

3. ***Traction*** (Gino Wickman)

   Finally, use EOS concepts to embed SOPs into organizational operations:

   - Clarify ownership via an accountability chart,
   - Ensure AppSec KPIs appear on scorecards,
   - Set Rocks for quarterly improvements,
   - Establish regular cadence for process review.

   Focus chapters: The Accountability Chart, Rocks, The Scorecard, The Issues List.



Navigation tip. If you already have mature processes and want to focus on specific improvements, you can skip directly to the blueprints (Section 5), SOPs (Section 6), or metrics framework (Section 7).

4 Modern AppSec Landscape

Before diving into process modeling, it's essential to understand the current AppSec tooling landscape and threat environment. Modern AppSec has evolved significantly beyond traditional SAST/DAST.

4.1 The Modern Threat Landscape

Contemporary AppSec programs must address:

- **Supply chain attacks**: Compromised dependencies, typosquatting, build system attacks (e.g., SolarWinds, Log4Shell, XZ Utils).
- **Container and cloud-native risks**: Misconfigured containers, vulnerable base images, Kubernetes security.
- **Infrastructure as Code (IaC) vulnerabilities**: Misconfigured Terraform, CloudFormation, or Kubernetes manifests.
- **API security**: Broken authentication, authorization flaws, data exposure via APIs.
- **AI/ML security**: Prompt injection, model poisoning, insecure AI integrations, AI-generated code risks.
- **Secrets exposure**: Hardcoded credentials, leaked API keys, insufficient rotation.

4.2 Modern Tooling Categories

Category	Purpose	Example Tools
SAST	Static code analysis for vulnerabilities	Semgrep, CodeQL, SonarQube, Checkmarx
SCA	Dependency vulnerability scanning	Snyk, Dependabot, Renovate, Trivy
Secrets Scanning	Detect hardcoded secrets	GitLeaks, TruffleHog, GitHub Secret Scanning
Container Security	Image scanning, runtime protection	Trivy, Grype, Prisma Cloud, Falco
IaC Scanning	Infrastructure misconfigurations	Checkov, tfsec, KICS, Terrascan
DAST	Dynamic application testing	OWASP ZAP, Burp Suite, Nuclei
API Security	API-specific testing	42Crunch, Noname Security, Salt Security
SBOM	Software bill of materials	Syft, CycloneDX, SPDX
ASPM	Application Security Posture Management	Apiiro, Cycode, ArmorCode

Table 1: Modern AppSec tooling categories

4.3 Supply Chain Security

Supply chain security has become a critical focus area. Key practices include:

- **SBOM Generation**: Creating and maintaining Software Bills of Materials for all artifacts.
- **Dependency pinning**: Using lock files and hash verification.
- **SLSA Framework**: Implementing Supply-chain Levels for Software Artifacts.
- **Artifact signing**: Using Sigstore/Cosign for container image and artifact signing.
- **Provenance attestation**: Documenting build provenance for audit trails.
- **Private registries**: Controlling which packages can be consumed.

4.4 AI/ML Security Considerations

As AI becomes integrated into development workflows, new security considerations emerge:

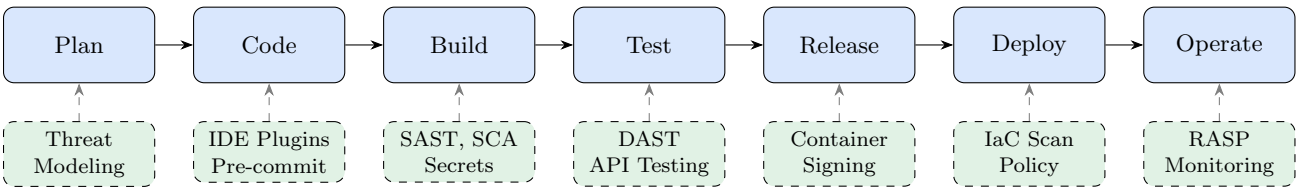
- **AI-generated code review**: Code from Copilot, Claude, or other assistants may contain vulnerabilities or insecure patterns.
- **Prompt injection**: Applications using LLMs may be vulnerable to prompt injection attacks.
- **Model security**: Protecting ML models from extraction, poisoning, and adversarial inputs.
- **Data leakage**: Ensuring sensitive data isn't exposed through AI training or inference.

Warning

AI-generated code should be treated with the same (or greater) scrutiny as human-written code. Security scanning and code review remain essential.

4.5 DevSecOps Integration Principles

Modern AppSec integrates throughout the SDLC:



Key DevSecOps principles:

- **Shift left**: Detect issues as early as possible in the SDLC.
- **Automate everything**: Security gates should be automated, not manual.

- **Developer enablement**: Provide clear guidance, not just findings.
- **Measure and iterate**: Use data to continuously improve processes.
- **Blameless culture**: Focus on fixing systems, not blaming people.

5 Modeling the AppSec Pipeline: Service Blueprints

This section provides service blueprints for key AppSec processes using the *This Is Service Design Doing* framework.

5.1 High-Level AppSec Pipeline Overview

At a high level, a modern AppSec pipeline includes:

1. Developer writes code (possibly with AI assistance) and opens a PR,
2. CI pipeline executes security scans (SAST, SCA, secrets, IaC, container),
3. Results surface on the PR and dashboards,
4. Developer addresses findings or requests exceptions,
5. Approved changes merge and deploy through progressive gates,
6. Critical findings feed into vulnerability management,
7. Continuous monitoring operates in production.

5.2 Blueprint 1: Secure Pull Request Flow

Customer (Developer) Actions

1. Create feature branch from main/trunk,
2. Implement changes (code, tests, configuration),
3. Run local security checks (IDE plugins, pre-commit hooks),
4. Commit and push to remote,
5. Open Pull Request against target branch,
6. Review security findings annotated on PR,
7. Address findings (fix, suppress with justification, or request exception),
8. Respond to code review feedback,
9. Merge when all checks pass and approvals obtained.

Frontstage (Visible to Developer) Actions

- CI/CD pipeline status checks on PR,
- Inline annotations from SAST/SCA tools,
- Security summary comment on PR,
- Links to documentation/remediation guidance,
- Dashboard views (security overview, vulnerability trends),
- Exception request workflow UI.

Backstage Actions

- CI jobs orchestrating security scans,
- SAST engines analyzing code changes,
- SCA scanning dependency manifests and lock files,
- Secrets scanning checking for credential exposure,
- Container scanning for modified Dockerfiles,
- IaC scanning for infrastructure changes,
- Result aggregation and deduplication,
- Notification dispatch (Slack, email, ticketing).

Supporting Processes and Systems

- CI/CD platform (GitHub Actions, GitLab CI, Jenkins),
- Security scanning tools (configured with custom rules),
- Repository settings (branch protection, required checks),
- Secrets management platform,
- Ticketing system integration (Jira, Linear),
- Documentation platform (Confluence, Notion).

Evidence and Artifacts

- PR history and audit trail,
- CI/CD logs with scan results,
- Security finding records (with timestamps, status),
- Exception records with justifications,
- Approval audit trail,
- SBOM for merged changes.

5.3 Blueprint 2: Vulnerability Triage and Management

Customer Actions (AppSec Analyst/Developer)

1. Receive notification of new vulnerability (from scanner or external advisory),
2. Review vulnerability details (CVSS, exploitability, affected systems),
3. Assess business context (data sensitivity, exposure, blast radius),
4. Assign severity and priority,
5. Route to appropriate team/owner,
6. Track remediation progress,
7. Verify fix and close finding.

Frontstage Actions

- Vulnerability dashboard with triage queue,
- Severity/priority assignment interface,
- Assignment and routing workflow,
- Status tracking and SLA indicators,
- Reporting and trends visualization.

Backstage Actions

- Vulnerability ingestion from multiple sources,
- Deduplication and correlation,
- Auto-enrichment with threat intelligence,

- SLA calculation and escalation triggers,
- Integration with ticketing systems,
- Metrics aggregation for reporting.

Supporting Processes

- Vulnerability database/platform,
- Threat intelligence feeds,
- Asset inventory (which repos/services exist),
- SLA definitions by severity,
- Escalation policies.

Evidence

- Triage records with justifications,
- Assignment and handoff history,
- Remediation timeline documentation,
- Verification evidence (scan results, test results),
- Exception/risk acceptance records.

5.4 Blueprint 3: Container Security

Customer (Developer) Actions

1. Create or modify Dockerfile/container configuration,
2. Build container image locally or in CI,
3. Review container scan results,
4. Address vulnerabilities (update base image, remove packages),
5. Push signed image to registry,
6. Deploy using approved image.

Frontstage Actions

- Container scan results in CI output,
- Registry UI showing image vulnerability status,
- Approved base image catalog,
- Deployment policy status.

Backstage Actions

- Image scanning in CI pipeline,
- Continuous registry scanning,
- Base image vulnerability monitoring,
- Image signing with Cosign/Sigstore,
- Admission controller policy enforcement,
- SBOM generation and storage.

Supporting Processes

- Container registry (with vulnerability scanning),
- Approved base image library,
- Kubernetes admission controller,
- SBOM storage and query system.

Evidence

- Image scan reports,
- Signing attestations,
- SBOM records,
- Deployment audit logs.

5.5 Blueprint 4: Secrets Incident Response

Trigger

Secret detected in code, logs, or external exposure (e.g., GitHub public exposure alert).

Customer (Responder) Actions

1. Receive alert of exposed secret,
2. Assess scope (what systems have access, what data is at risk),
3. Revoke/rotate the compromised credential immediately,
4. Audit access logs for unauthorized use,
5. Identify root cause (how did secret get exposed),
6. Implement preventive measures,
7. Document incident and close.

Frontstage Actions

- Incident alert with severity and affected secret type,
- Runbook links for specific secret types,
- Status tracking dashboard,
- Communication templates.

Backstage Actions

- Secret detection and alerting,
- Automatic ticket creation,
- Integration with secrets management for rotation,
- Log aggregation for audit,
- Paging/escalation if SLA breached.

Evidence

- Incident record with timeline,
- Rotation confirmation,
- Access audit results,
- Root cause analysis,
- Remediation verification.

6 Standard Operating Procedures (SOPs)

This section provides *Work the System*-style SOPs for key AppSec processes. Each SOP follows a consistent structure: Purpose, Scope, Owner, Procedure, Exception Handling, Evidence, and Improvement Loop.

6.1 SOP: Secure Pull Request Flow

Purpose Ensure all code changes merged to protected branches have passed appropriate security checks and that exceptions are handled consistently with documented justification.

Scope All repositories using the standard CI/CD pipeline with security scanning enabled.

Owner AppSec Lead (process owner); Platform Engineering (tooling owner).

Procedure (Standard Flow)

1. Developer creates feature branch from the target branch.
2. Developer implements changes, including appropriate tests.
3. Developer runs local security checks (IDE plugins, pre-commit hooks) before pushing.
4. Developer opens Pull Request to the target branch.
5. CI pipeline triggers automatically and executes:
   - Unit/integration tests,
   - SAST scan (e.g., Semgrep, CodeQL),
   - SCA scan (e.g., Dependabot, Snyk),
   - Secrets scan,
   - Container scan (if Dockerfile modified),
   - IaC scan (if infrastructure files modified).
6. Results annotate the PR with findings and severity.
7. If all required checks pass with no blocking findings:
   - Developer obtains required code review approvals,
   - PR is eligible for merge.
8. If blocking findings exist, proceed to Exception Handling.

Exception Handling

1. Developer investigates finding and attempts remediation.
2. If remediation is not feasible within acceptable timeframe:
   - Developer creates exception request with: finding details, business justification, risk assessment, proposed mitigation, and timeline for proper fix.
3. Exception request is reviewed by AppSec (and security champion if applicable).
4. AppSec either:
   - Approves with conditions (maximum duration, required mitigations),
   - Rejects with guidance for remediation.
5. Approved exceptions are logged with expiration date and tracked to resolution.

Evidence and Records

- Scan results attached to PR,
- CI/CD logs,
- Exception request records with approval/rejection and justification,
- Audit trail of merges and approvals.

Improvement Loop

- Monthly: Review aggregate scan data, identify false positive patterns, tune rules.
- Quarterly: Review exception trends, identify systemic issues, update SOP as needed.
- Annually: Full process review with developer feedback.

6.2 SOP: Vulnerability Triage

Purpose Ensure newly identified vulnerabilities are assessed, prioritized, and routed for remediation consistently based on risk.

Scope All vulnerabilities from automated scanners, penetration tests, bug bounty, and external advisories.

Owner AppSec Lead (process); Vulnerability Management Analyst (execution).

Procedure

1. New vulnerability enters triage queue (from scanner, advisory, or manual report).
2. Analyst reviews within SLA (Critical: 4 hours, High: 1 business day, Medium/Low: 3 business days).
3. Analyst assesses:
   - Technical severity (CVSS base score),
   - Exploitability (is exploit available? is it being exploited in the wild?),
   - Business context (data sensitivity, internet exposure, blast radius),
   - Existing mitigations (WAF rules, network controls).
4. Analyst assigns adjusted priority:
   - P1/Critical: Immediate action required, SLA 24–72 hours.
   - P2/High: Urgent, SLA 7–14 days.
   - P3/Medium: Standard, SLA 30–60 days.
   - P4/Low: Backlog, address opportunistically.
5. Analyst routes to responsible team via ticketing system.
6. Analyst sets SLA deadline and escalation triggers.
7. Responsible team remediates and marks complete.
8. Analyst verifies fix (re-scan or manual verification) and closes finding.

Exception/Risk Acceptance

1. If team cannot meet SLA, they request risk acceptance.
2. Risk acceptance requires: business justification, compensating controls, owner signature (director+ for P1/P2).
3. Risk acceptances expire and must be renewed or resolved.

Evidence

- Triage record with assessment rationale,
- Assignment and routing history,
- Remediation evidence,
- Risk acceptance documentation.

Improvement Loop

- Weekly: Review SLA compliance and escalations.
- Monthly: Analyze root causes of recurring vulnerability types.
- Quarterly: Review risk acceptance backlog, refine prioritization criteria.

6.3 SOP: Secrets Incident Response

Purpose Ensure exposed secrets are revoked rapidly and the incident is fully investigated to prevent recurrence.

Scope All detected secret exposures in code, logs, configuration, or external reports.

Owner AppSec Lead (process); On-call security engineer (execution).

Procedure

1. Alert received (automated detection or external report).
2. Responder acknowledges within 15 minutes for production secrets.
3. Responder assesses scope:
   - What type of secret (API key, database credential, signing key)?
   - What systems/data could be accessed?
   - Was the exposure public or internal-only?
   - How long was it exposed?
4. Responder immediately revokes/rotates the credential:
   - Coordinate with service owner if rotation causes outage.
   - If rotation not immediately possible, implement compensating controls.
5. Responder audits access logs for unauthorized usage.
6. If unauthorized access confirmed, escalate to full incident response.
7. Responder identifies root cause:
   - How did the secret get committed/exposed?
   - Why didn't existing controls catch it?
8. Responder implements preventive measures:
   - Update .gitignore or pre-commit hooks,

- Improve secret scanning rules,
- Developer education if needed.

9. Responder documents incident and closes with lessons learned.

Evidence

- Incident ticket with timeline,
- Rotation confirmation,
- Access audit results,
- Root cause analysis,
- Preventive measures implemented.

SLAs

- Production secrets (public exposure): Rotate within 1 hour.
- Production secrets (internal exposure): Rotate within 4 hours.
- Non-production secrets: Rotate within 24 hours.

6.4 SOP: Third-Party Dependency Approval

Purpose Ensure new dependencies are evaluated for security risk before adoption.

Scope New dependencies not previously approved, or major version upgrades with significant changes.

Owner AppSec Lead (approval authority); Developers (requesters).

Procedure

1. Developer identifies need for new dependency.
2. Developer checks approved dependency list; if already approved, proceed with usage.
3. If not approved, developer submits dependency review request:
   - Package name, version, registry (npm, PyPI, etc.),
   - Business justification,
   - Alternatives considered.

4. AppSec reviews within 3 business days:
    - Maintenance status (last update, maintainer activity),
    - Known vulnerabilities,
    - License compatibility,
    - Dependency tree analysis,
    - Security posture (does it follow secure development practices?).
5. AppSec decision:
    - Approved: Added to approved list,
    - Approved with conditions: Usage restrictions documented,
    - Rejected: Alternative recommended.
6. Approved dependencies monitored for new vulnerabilities.

Evidence

- Review request and decision record,
- Approved dependency inventory,
- Condition/restriction documentation.

7 Metrics Framework

Effective AppSec programs require metrics that drive behavior and demonstrate value. This section provides a framework aligned with *Traction*'s scorecard concept.

7.1 Metric Categories

Category	Purpose	Example Metrics
Coverage	Are security controls deployed?	% repos with SAST, % images scanned
Effectiveness	Are controls finding issues?	True positive rate, findings per scan
Remediation	Are issues being fixed?	MTTR by severity, SLA compliance
Risk Posture	What is current exposure?	Open criticals, vulnerability age distribution
Developer Experience	Is security enabling or blocking?	Friction score, false positive rate
Program Health	Is the program maturing?	SOP adherence, training completion

Table 2: AppSec metric categories

7.2 Recommended Scorecard Metrics

For weekly/monthly leadership scorecard:

1. **Open Critical Vulnerabilities**: Count of unresolved P1/Critical findings. Target: 0.
2. **Critical MTTR**: Mean time to remediate critical findings. Target: < 72 hours.
3. **High MTTR**: Mean time to remediate high findings. Target: < 14 days.
4. **SLA Compliance Rate**: Percentage of findings resolved within SLA. Target: $> 95\%$.
5. **Scanning Coverage**: Percentage of production repos with security scanning enabled. Target: 100%.
6. **Security Gate Pass Rate**: Percentage of PRs passing security checks on first attempt. Target: $> 90\%$.
7. **Exception Backlog**: Count of active security exceptions. Target: Declining trend.
8. **False Positive Rate**: Percentage of findings marked as false positive. Target: $< 10\%$.

7.3 Leading vs. Lagging Indicators

- **Leading indicators** predict future security posture:
  - Security training completion rate,

  - Scanning coverage expansion,
  - Security champion engagement,
  - Pre-commit hook adoption.

- **Lagging indicators** measure past performance:
  - Vulnerabilities discovered in production,
  - Security incidents,
  - MTTR metrics,
  - Audit findings.

A healthy program tracks both.

7.4 Metric Anti-Patterns

Avoid metrics that drive wrong behavior:

- **Raw finding counts**: Incentivizes suppressing findings rather than fixing them.
- **Vulnerabilities found**: Rewards noisy tools over accurate ones.
- **Tickets closed**: Encourages closing without proper verification.

Key Insight

Good metrics align incentives with outcomes. Measure what matters (risk reduction, developer productivity) not what's easy to count (tickets, findings).

8 Alignment with Industry Maturity Models

Your AppSec process modeling efforts should align with industry-recognized maturity models. This provides external validation and a roadmap for improvement.

8.1 OWASP SAMM

The OWASP Software Assurance Maturity Model (SAMM) provides a framework for assessing and improving software security programs across five business functions:

1. **Governance**: Strategy, policy, compliance, education.
2. **Design**: Threat assessment, security requirements, security architecture.
3. **Implementation**: Secure build, secure deployment, defect management.
4. **Verification**: Architecture assessment, requirements-driven testing, security testing.
5. **Operations**: Incident management, environment management, operational management.

Each function has three maturity levels. Your SOPs and blueprints map directly to SAMM practices.

8.2 BSIMM

The Building Security In Maturity Model (BSIMM) is descriptive rather than prescriptive—it measures what organizations actually do. Key domains include:

- Governance,
- Intelligence (attack models, security features, standards),
- SSDL Touchpoints (architecture analysis, code review, security testing),
- Deployment (penetration testing, environment, operations).

BSIMM assessments benchmark your program against peers.

8.3 NIST Cybersecurity Framework

The NIST CSF provides high-level functions applicable to AppSec:

- **Identify**: Asset inventory, risk assessment.
- **Protect**: Secure development, access control.
- **Detect**: Scanning, monitoring.
- **Respond**: Incident response, communications.
- **Recover**: Recovery planning, improvements.

8.4 Mapping Your Program

Your Artifact	SAMM Mapping	NIST CSF Mapping
Secure PR SOP	Implementation: Secure Build	Protect: PR.DS
Vulnerability Triage SOP	Implementation: Defect Management	Detect, Respond
Secrets Incident SOP	Operations: Incident Management	Respond, Recover
Metrics Framework	Governance: Strategy & Metrics	Identify
Security Champion Program	Governance: Education	Protect

Table 3: Mapping artifacts to maturity models

9 Putting It All Together: Implementation Roadmap

This section synthesizes the frameworks, blueprints, and SOPs into a practical implementation roadmap.

9.1 Phase 1: Foundation (Weeks 1–4)

1. **Assess current state**:
   - Inventory existing security tools and their coverage,
   - Map current (often undocumented) processes,
   - Identify key pain points from developer feedback.
2. **Create developer journey map**:
   - Document how developers experience security today,
   - Identify friction points and gaps.
3. **Draft Strategic Objective and Operating Principles**:
   - What does "good" look like for your AppSec program?
   - What principles guide decisions?

9.2 Phase 2: Documentation (Weeks 5–8)

1. **Create 2–3 service blueprints** for highest-impact flows:
   - Secure PR Flow (almost always the starting point),
   - Vulnerability Triage,
   - One other based on your context (Container Security, Secrets Management).
2. **Write corresponding SOPs**:
   - Follow the structure in Section 6,
   - Validate with practitioners—does this match reality?
3. **Define initial metrics**:
   - Start with 3–5 metrics from Section 7,
   - Establish baselines.

9.3 Phase 3: Organizational Integration (Weeks 9–12)

1. **Create accountability chart**:
   - Assign explicit owners to each SOP and blueprint,
   - Ensure no orphaned processes.
2. **Set initial Rocks** (90-day priorities):
   - Pick 2–3 measurable improvements (e.g., "Reduce MTTR for criticals by 50%").
3. **Integrate into leadership cadence**:
   - Add AppSec metrics to executive scorecard,
   - Schedule quarterly AppSec review.
4. **Launch security champion program** (if not existing):
   - Recruit 1 champion per team,
   - Provide training and recognition.

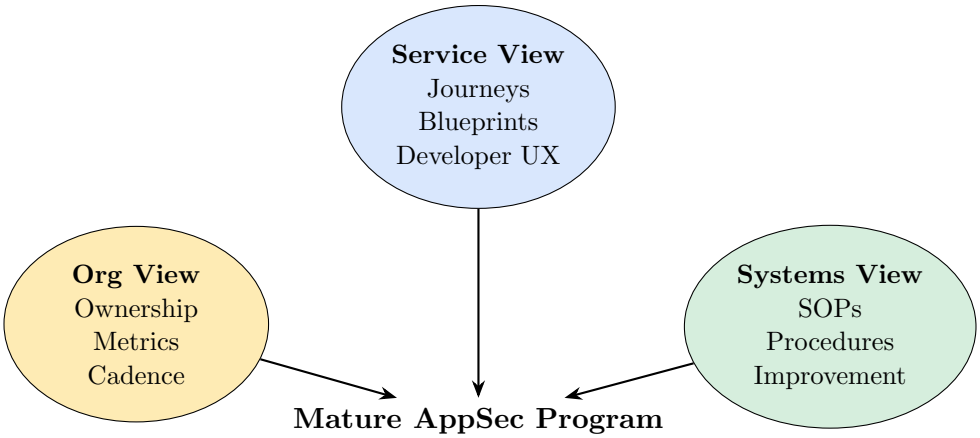
9.4 Phase 4: Continuous Improvement (Ongoing)

1. **Monthly**: Review metrics, tune scanning rules, address false positives.
2. **Quarterly**: Review SOPs, update based on lessons learned, refresh Rocks.
3. **Annually**: Full maturity assessment (SAMM or BSIMM), strategic planning.

9.5 The Combined Model

The three books combine to provide:

- **Service view** (*This Is Service Design Doing*): Developer experience, journeys, touchpoints, support channels.
- **Systems view** (*Work the System*): Documented SOPs, defined inputs/outputs, incremental improvement.
- **Organizational view** (*Traction*): Ownership, metrics, cadence, accountability.



Done well, this transforms AppSec from a loose collection of tools and gates into a coherent, documented system that is easier to operate, explain, measure, and improve.

A Templates

A.1 SOP Template

SOP Template

SOP Title: [Process Name]
Version: [X.Y] **Last Updated:** [Date]
Owner: [Name/Role]

Purpose
[One paragraph describing why this process exists and what outcome it ensures.]

Scope
[What systems, teams, or situations does this apply to?]

Procedure

1. [Step 1]
2. [Step 2]
3. [Continue...]

Exception Handling
[How are deviations from the standard process handled?]

Evidence and Records
[What artifacts document this process was followed?]

SLAs
[Time-bound commitments, if applicable.]

Improvement Loop
[How often is this SOP reviewed? By whom?]

A.2 Exception Request Template

Security Exception Request

Requester: [Name] **Date:** [Date]
Team/Service: [Affected service]

Finding Details

- Finding ID: [Scanner finding ID]
- Severity: [Critical/High/Medium/Low]
- Tool: [Which scanner found this]
- Description: [Brief description]

Business Justification
[Why can't this be fixed within standard SLA?]

Risk Assessment
[What is the realistic risk of this vulnerability being exploited?]

Compensating Controls
[What mitigations are in place to reduce risk?]

Remediation Plan
[How and when will this be properly fixed?]
Proposed exception duration: [X days/weeks]

Approval
AppSec Review: _____ Date: _____
Decision: ☐ Approved ☐ Approved with conditions ☐ Rejected

A.3 Service Blueprint Template

Service Blueprint Template

Blueprint: [Process Name]
Version: [X.Y] **Last Updated:** [Date]

Customer Actions (What the developer/user does)

- [Action 1]
- [Action 2]

Frontstage Actions (What the user sees/interacts with)

- [Visible element 1]
- [Visible element 2]

Backstage Actions (What happens behind the scenes)

- [Backend process 1]
- [Backend process 2]

Supporting Processes and Systems

- [System 1]
- [System 2]

Evidence and Artifacts

- [Artifact 1]
- [Artifact 2]

B Modern Tooling Quick Reference

Tool	Category	Key Features
Static Analysis (SAST)		
Semgrep	SAST	Fast, customizable rules, supports 30+ languages
CodeQL	SAST	GitHub-native, semantic analysis, extensive query library
SonarQube	SAST	Broad language support, quality gates, enterprise features
Software Composition Analysis (SCA)		
Snyk	SCA	Developer-friendly, fix suggestions, broad ecosystem
Dependabot	SCA	GitHub-native, automated PRs for updates
Renovate	SCA	Highly configurable, multi-platform
Secrets Scanning		
GitLeaks	Secrets	Fast, regex and entropy detection
TruffleHog	Secrets	Deep history scanning, verified secrets
GitHub Secret Scanning	Secrets	Native integration, partner program for revocation
Container Security		
Trivy	Container/SCA	Comprehensive scanner, OS and language packages
Grype	Container/SCA	Fast, SBOM-native
Prisma Cloud	Container	Enterprise, runtime protection
IaC Security		
Checkov	IaC	Multi-framework, 1000+ policies
tfsec	IaC	Terraform-focused, fast
KICS	IaC	Broad IaC support, extensible
SBOM & Supply Chain		
Syft	SBOM	Generate SBOMs from containers/filesystems
Cosign	Signing	Container image signing, keyless support
SLSA	Framework	Supply chain integrity levels

Table 4: Modern AppSec tooling reference

B.1 Tool Selection Criteria

When evaluating tools, consider:

- **Accuracy**: Low false positive rate, high true positive rate.
- **Developer experience**: Clear findings, actionable guidance, IDE integration.
- **Integration**: CI/CD compatibility, API availability, webhook support.
- **Customization**: Can you write custom rules for your context?
- **Performance**: Scan time acceptable for CI/CD?
- **Coverage**: Languages, frameworks, and package ecosystems you use.
- **Total cost**: License cost, maintenance burden, training investment.

Document History

Version	Date	Changes
1.0	Initial	Original document
2.0	November 30, 2025	Comprehensive update: Added modern tooling landscape, supply chain security, AI/ML considerations, additional blueprints (Container Security, Secrets Incident Response), expanded SOPs, metrics framework, maturity model alignment, implementation roadmap, templates, and tooling reference.