Stoked: Stochastic Typed Operations Kit for Engineering Delivery

A Formal Specification Language for Prompt-Based Production Systems with Dual Petri Net and Queueing Semantics

The Stoked Project

February 2026 — Version 0.1.0 (Draft)

Abstract

We introduce Stoked (**S**tochastic **T**yped **O**perations **K**it for **E**ngineering **D**elivery), a formal specification language for describing production systems in which workstations are LLM-based agents, deterministic computations, or human task queues, and jobs are software artifacts flowing through a queueing network. Stoked provides a unified formal model: every program has both a *control-flow* interpretation via Coloured Generalized Stochastic Petri Nets [**?** **?**] and a *performance* interpretation via queueing network extraction [**?**]. The language inherits compositional reasoning from CSP [**?**] and the π-calculus [**?**], while its performance semantics draws on Jackson networks [**?**], the BCMP theorem [**?**], and the VUT equation from Factory Physics [**?**]. We present the complete abstract syntax, type system, operational semantics, dual denotational translations, well-formedness conditions, and three worked examples drawn from the software lifecycle.

Contents

1 Introduction

Modern distributed software systems are *production systems* in the Operations Research and Industrial Engineering (ORIE) sense. Artifacts—code commits, pull requests, test reports, deployment manifests, incident tickets—flow through a network of workstations (development, review, testing, deployment, monitoring), are transformed by servers (human engineers, automated tools, LLM-based agents), experience congestion and variability, and must meet throughput and cycle-time targets.

Yet the tools used to specify, analyze, and orchestrate these systems lack the formal foundation that manufacturing and logistics have enjoyed for decades. Agent orchestration frameworks describe *what* to do but not *how fast* or *how reliably.* Workflow engines capture sequencing but ignore queueing effects. DevOps pipelines are operationally concrete but analytically opaque.

Stoked bridges this gap. It is a formal specification language for compiling high-level descriptions of productive processes into graphs of prompt-based processes for developing, deploying, monitoring, and maintaining distributed software systems. Crucially, Stoked is *not* a generic agent orchestration DSL. It is a **production system specification language** grounded in ORIE, where:

- **Workstations** are prompt-based LLM agents, deterministic compute steps, or human task queues.
- **Jobs** are software artifacts.
- **The graph** is a queueing network with formal performance semantics.

1.1 Design Principles

Stoked rests on five principles:

1. **Dual semantics by construction.** Every Stoked program has both a control-flow interpretation (what can happen) via Coloured Generalized Stochastic Petri Nets [**?**], and a performance interpretation (how fast it happens) via queueing network extraction. These are not separate models glued together; they arise from a single unified formal model.
2. **Stochastic from day one.** Variability is not an afterthought. Arrival processes, service times, yield rates, and routing probabilities are first-class language constructs with distribution types. The Squared Coefficient of Variation ($c^2 = \text{Var}/\text{Mean}^2$) links the language's stochastic specifications directly to queueing performance via the VUT equation [**? ?**].
3. **Process-algebraic composition.** Stoked inherits the compositional reasoning power of CSP [**? ?**] and the π-calculus [**? ? ?**]. Systems are built from primitive processes using sequential, parallel, choice, restriction, and replication operators. Every operator has a precise Petri net translation and queueing interpretation.
4. **ORIE-native abstractions.** The language provides first-class constructs for Factory

Physics [?] concepts: stations with service disciplines, finite resources (WIP caps, Kanban), arrival processes, routing policies, batch processing, rework loops, and performance assertions.

5. **Specification, not implementation.** Stoked defines syntax, type system, and formal semantics. Conforming implementations may target simulation, analytical solvers, model checkers, or direct execution.

1.2 Contributions

This paper presents:

1. A complete abstract syntax (EBNF grammar) for a process algebra enriched with ORIE primitives (Section 3).
2. A type system with kinds, distribution types, resource environments, and dimensional types (Section 4).
3. Structural operational semantics in the style of Plotkin [?] with stochastic extensions (Section 5).
4. A translation $[\![\cdot]\!]$ to Coloured Generalized Stochastic Petri Nets [? ?] with a behavioral equivalence theorem (Section 6).
5. A queueing model extraction function $\mathcal{Q}(\cdot)$ supporting Jackson [?], BCMP [?], and GI/G/c approximations (Section 7).
6. Eight well-formedness conditions bridging type safety, structural analysis, and performance analysis (Section 8).
7. Three worked examples from the software lifecycle (Section 9).

1.3 Paper Organization

Section 2 reviews the formal foundations. Section 3 presents the abstract syntax. Section 4 defines the type system. Section 5 gives the operational semantics. Section 6 defines the Petri net translation. Section 7 describes the queueing extraction. Section 8 states the well-formedness conditions. Section 9 presents worked examples. Section 10 discusses related work, and Section 11 concludes. The full specification (12 chapters, ∼4,600 lines) accompanies this paper.

2 Background and Foundations

2.1 Process Algebra

Stoked draws on two traditions of process algebra.

CSP (Communicating Sequential Processes) [? ?] provides the alphabetized model of synchronization, external and internal choice operators, and the trace/failures/divergences semantic hierarchy. A CSP process communicates with its environment through named events; parallel composition synchronizes on shared alphabets.

The π-**calculus** [? ? ?] extends this with *name mobility*: channels are first-class values that can be communicated over other channels. The restriction operator $(\nu\, a)P$ creates fresh private channels, and replication $!P$ models persistent servers. STOKED adopts typed channels from the π-calculus with asymmetric send/receive, and restriction for scope control.

2.2 Petri Nets

Petri nets [? ? ?] are a classical model for concurrent and distributed systems. A Petri net (P, T, F, W, M_0) consists of places P, transitions T, a flow relation F, arc weights W, and an initial marking M_0. The theory provides powerful structural analysis: *P-invariants* for conservation, *T-invariants* for repeatability, *siphons* and *traps* for deadlock analysis.

STOKED targets *Coloured Generalized Stochastic Petri Nets* (CGSPNs) [? ?], which extend the basic model with token colors (types), guards, arc expressions, and stochastic firing delays. Channels map to places, stations map to transition subnets, and composition operators map to net composition operations.

2.3 Queueing Theory

The queueing-theoretic grounding of STOKED draws on four pillars:

Jackson networks [?]: Open networks of $M/M/c$ queues with Poisson arrivals and exponential service admit product-form steady-state solutions, enabling efficient analytical computation.

The BCMP theorem [?]: Generalizes product-form results to four server types (FCFS with exponential service, processor sharing, infinite server, LCFS-PR) with multiple job classes and state-dependent routing.

Little's Law [? ?]: The universal invariant $L = \lambda W$ relating average number in system, throughput, and average sojourn time. It holds for *any* stable system regardless of distributional assumptions.

The VUT equation [? ?]: For a GI/G/c queue, the expected waiting time in queue is approximately:

$$W_q \approx \underbrace{\left(\frac{c_a^2 + c_s^2}{2}\right)}_{\text{Variability}} \cdot \underbrace{\frac{\rho^{\sqrt{2(c+1)}}}{c\,(1-\rho)}}_{\text{Utilization}} \cdot \underbrace{E[S]}_{\text{Time}} \tag{1}$$

where c_a^2 and c_s^2 are the squared coefficients of variation of inter-arrival and service times, $\rho = \lambda/(c\mu)$ is utilization, and $E[S]$ is mean service time. This connects distribution parameters directly to performance, which is why STOKED makes distributions first-class.

2.4 Factory Physics

The ORIE grounding draws on the laws of Factory Physics [**?**]: *Little's Law* (WIP $=$ TH$\times$CT), the *VUT equation* (Eq. 1), *bottleneck rate* r_b and *raw process time* T_0, *critical WIP* $W_0 = r_b \cdot T_0$, and the *practical worst case* $\mathrm{CT}_{\mathrm{pwc}} = T_0 + (W-1)/r_b$. These provide the analytical framework for performance assertions.

3 Abstract Syntax

The complete EBNF grammar (∼540 lines) is given in the accompanying specification (Chapter 3). Here we highlight the key design decisions.

3.1 Five Primitive Declarations

Stoked has exactly five top-level declaration forms, each corresponding to an ORIE concept, a Petri net element, and a queueing-theoretic entity:

Declaration	**ORIE Concept**	**Petri Net**	**Queueing**
`type`	Job/work item type	Token color set	Job class
`channel`	Queue/buffer	Place	Queue
`station`	Workstation/server	Transition subnet	Service center
`resource`	Shared finite resource	Resource place	—
`arrival`	Arrival process	Source transition	External arrival

Table 1: The five primitive declarations and their interpretations.

3.2 Process Algebra Operators

Processes are composed using operators from CSP and the π-calculus:

Syntax	Origin	Description
`P ; Q`	CSP	Sequential composition
`P | Q`	CSP/π	Synchronized parallel
`P ||| Q`	CSP	Interleaved parallel
`P |[S]| Q`	CSP	Alphabetized parallel (sync on S)
`P [] Q`	CSP	External choice
`P |~| Q`	CSP	Internal choice
`pchoice`	New	Probabilistic choice
`a ! v / a ? x`	π	Async send/receive
`a !! v / a ?? x`	π	Sync send/receive (rendezvous)
$(\nu\, a : \mathrm{Chan}\langle T\rangle)\, P$	π	Restriction (new channel)
`!P`	π	Replication
`delay(D)`	New	Timed delay (distribution D)

Table 2: Process algebra operators.

3.3 Station Service Types

Each station abstracts over one of three service mechanisms:

- `prompt { ... }` — LLM-based agent (stochastic, configurable model/temperature/tools).
- `compute { ... }` — Deterministic computation.
- `human { ... }` — Human task (with SLA, escalation, schedule constraints).

All three share the same queueing semantics: a service center with a given number of servers, a service time distribution, and a queue discipline. The distinction matters only for the *implementation* of the service process, not for the *performance model*.

3.4 First-Class Distributions

Distributions appear in five positions: arrival rates, service times, yield/quality, routing probabilities, and stochastic let-bindings. The standard library provides 15 continuous and discrete distributions plus 7 combinators (mixture, truncation, shift, scale, max, min, convolution).

Every distribution D has a known mean $E[D]$, variance $\mathrm{Var}[D]$, and SCV $c^2(D) = \mathrm{Var}[D]/E[D]^2$. The SCV is the key link to queueing performance via Equation 1.

4 Type System

STOKED has a kind system classifying types into `Type` (values), `Dist` (distributions), `Proc` (processes), `Chan` (channels), and `Res` (resources).

The type universe has four layers: base types (Bool, Int, Float, String, Unit, Duration, Rate), composite types (records, variants, lists, sets, maps, tuples, options), domain-specific types

(Chan$\langle T\rangle$, Dist$\langle T\rangle$, Resource$\langle n\rangle$), and the process type `proc`.

Duration and **Rate** are dimensioned types enforcing dimensional consistency: multiplying a Rate by a Duration yields a Float (dimensionless), enabling type-safe Little's Law computations.

Channel subtyping is *invariant* to prevent type errors in concurrent communication. Record subtyping is structural (width and depth).

Process typing uses a *resource environment* Δ that tracks acquired resource units, with a merge operator $\oplus$ that ensures no over-allocation across parallel components. The typing judgment $\Gamma; \Delta \vdash P : \texttt{proc}$ ensures both type correctness and resource safety.

5 Operational Semantics

The operational semantics follows Plotkin's structural approach [**?**]. A *configuration* $C = \langle P, \sigma, \beta, \rho, t\rangle$ comprises a process P, a value store σ, a buffer state β (mapping channels to queues), a resource state ρ (available units per resource), and a global clock t.

Labels classify actions: internal (τ), channel send/receive, timed delay, resource acquire/release, station firing, sampling, and arrival.

Structural congruence establishes that parallel composition is commutative and associative, sequential composition has unit `skip`, and replication unfolds: $!P \equiv P \,|!P$.

The specification provides reduction rules for all process constructors: sequential composition, async and sync communication, station firing (with yield/rework), resource acquisition/release, all three parallel compositions, all three choice operators, time passage, restriction, replication, let-binding, conditionals, pattern matching, recursion, station invocation, and monitor expressions.

Maximal progress: Internal actions (τ) take priority over time passage. Time advances only when no τ-actions are enabled.

Stochastic extension: When all distributions are exponential, the semantics induces a Continuous-Time Markov Chain (CTMC). For general distributions, it induces a Generalized Semi-Markov Process (GSMP) with race-condition event selection.

6 Petri Net Semantics

The translation function $[\![\cdot]\!]$ maps Stoked programs to Coloured Generalized Stochastic Petri Nets (CGSPNs) [**?**].

6.1 Translation Scheme

The key correspondences are:

- **Channels** $\rightarrow$ **Places**: A channel `a : Chan<T>` maps to a place p_a with color set $[\![T]\!]$.

- **Stations → Transition subnets**: A station with c servers maps to a three-place, two-transition subnet modeling the $M/G/c$ queueing behavior: an idle-server place (initial marking c), a busy-server place, and start/done transitions.
- **Resources → Resource places**: A resource with capacity n maps to a place with initial marking n. Acquisition/release arcs connect to station transitions.
- **Arrivals → Source transitions**: Self-enabling timed transitions.
- **Composition operators → Net composition**: Sequential composition merges terminal/initial places; parallel composition fuses shared channel places; alphabetized parallel fuses only the synchronization set.

6.2 Behavioral Equivalence

Theorem 6.1 (Behavioral Equivalence)**.** *For every well-typed* Stoked *process P:*

$$\mathit{traces}(\mathit{SOS}(P)) = \mathit{traces}(\mathit{PN}(\llbracket P \rrbracket))$$

The proof proceeds by structural induction, establishing a weak bisimulation between SOS configurations and Petri net markings.

6.3 Structural Analysis

P-invariants of $\llbracket P \rrbracket$ yield conservation properties: server conservation (idle+busy $= c$), resource conservation, and flow balance. Siphons and traps provide sufficient conditions for deadlock-freedom checkable in polynomial time [**?** **?**].

7 Queueing Semantics

The extraction function $\mathcal{Q}(\cdot)$ maps Stoked systems to queueing network models.

7.1 Station Classification

Each station is classified by the BCMP theorem [**?**] into one of four types based on its discipline and service distribution (Table 3).

BCMP Type	Discipline	Service	Stoked Config
Type 1	FCFS	Exponential	`fifo`, `Exponential`
Type 2	PS	General	`ps`
Type 3	IS	General (delay)	`is`∞
Type 4	LCFS-PR	General	`lifo`, preemptive

Table 3: BCMP station type classification.

7.2 Analysis Methods

STOKED supports three analysis regimes:

1. **Jackson networks** [?] for open networks with Poisson arrivals and exponential service.
2. **Mean Value Analysis** [?] for closed networks.
3. **VUT approximation** [? ?] for general GI/G/c stations, using the departure approximation to propagate variability through the network.

Little's Law [?] serves as a universal invariant: $L = \lambda W$ at every level of the system (system-wide, per-station, per-queue).

7.3 Performance Assertions

Performance assertions connect the queueing model to engineering requirements:

```
assert throughput(System) >= 10/d
assert cycle_time(System).p95 <= 2d
assert utilization(Station) <= 0.85
assert bottleneck(System) == CodeReview
assert littles_law(System)
```

8 Well-Formedness Conditions

A STOKED system is *well-formed* if it satisfies eight conditions:

#	Condition	Ensures	Method
WF-1	Type safety	Preservation + progress	Type checking
WF-2	Deadlock-freedom	No stuck states	Siphon/trap analysis
WF-3	Boundedness	No unbounded buffer growth	P-invariant analysis
WF-4	Conservation	Flow balance at every station	P-invariant analysis
WF-5	Routing completeness	Every job reaches exit	Graph reachability
WF-6	Resource sufficiency	No starvation	Integer programming
WF-7	Stability	$\rho < 1$ at every station	Traffic equations
WF-8	SPC well-formedness	Valid monitoring constraints	Constraint checking

Table 4: The eight well-formedness conditions.

The conditions bridge three analysis domains: the type system (WF-1), the Petri net structure (WF-2 through WF-6), and the queueing model (WF-7). A system satisfying all eight conditions is guaranteed to be type-safe, deadlock-free, bounded, conservative, complete, resource-sufficient, stable, and monitorable.

9 Worked Examples

The specification includes three complete examples. We summarize them here.

9.1 CI/CD Pipeline with Code Review

A 4-station pipeline (Build $\rightarrow$ Code Review $\rightarrow$ Test $\rightarrow$ Deploy) with a rework loop (30% of reviews request changes). PRs arrive at 10/day. Traffic equations yield effective arrival rates accounting for rework: $\lambda_{\text{CodeReview}} = 10/(1 - 0.3) = 14.29$/day. The bottleneck is Code Review ($\rho = 0.149$). Performance assertions verify throughput ≥ 9.5/day, p95 cycle time ≤ 2 days, and Little's Law.

9.2 Incident Response System

A multi-path system with severity-based routing: alerts are triaged (AI-assisted), then routed to critical response (human, 24/7), auto-remediation (80% success), or logging. Failed auto-remediations escalate to human engineers. SPC monitoring detects response-time trends and throughput degradation. The system processes 100 alerts/day with p99 critical response time under 15 minutes.

9.3 Multi-Team Kanban Feature Delivery

A 3-team (Frontend, Backend, Platform) Kanban system with WIP limits, parallel development, integration testing with 25% rework, QA, and release. Features arrive at 3/week. The join (synchronization) point introduces a delay equal to $\max(D_{\text{FE}}, D_{\text{BE}}, D_{\text{Platform}})$. Critical WIP $W_0 \approx 3$ features; actual expected WIP $\approx$ 3–5, well within the WIP limit of 15.

10 Related Work

Process algebras for performance. Stochastic extensions of process algebras have been explored extensively. PEPA (Performance Evaluation Process Algebra) [?] associates rates with activities, yielding a CTMC. Stochastic π-calculus variants [?] add rates to the π-calculus. STOKED differs by targeting the full generality of GI/G/c queueing (not just exponential/CTMC), by providing first-class ORIE abstractions (stations, WIP limits, rework), and by maintaining a dual Petri net structure for analysis.

Workflow and orchestration languages. BPMN, YAWL, and temporal workflow specifications describe business processes but lack queueing semantics. STOKED occupies a different niche: it is a *formal specification* language, not an executable workflow engine.

Queueing network tools. Tools like LQNS, JINQS, and the JMT suite [?] solve queueing network models but are not compositional specification languages. STOKED provides a language-level abstraction that *generates* the queueing models to be solved.

Petri nets for software processes. The use of Petri nets for modeling software processes dates to [?]. STOKED contributes a *typed, compositional* process algebra whose Petri net translation preserves behavioral equivalence.

11 Conclusion and Future Work

We have presented Stoked, a formal specification language for prompt-based production systems with dual Petri net and queueing semantics. The language's five primitive declarations, process-algebraic operators, and first-class distributions provide a compositional framework for specifying, analyzing, and reasoning about the performance of modern software delivery systems.

Future directions include:

- A reference implementation (parser, type checker, Petri net translator, queueing solver).
- Model checking integration for automated verification of well-formedness conditions.
- Simulation backend for systems with general (non-exponential) distributions.
- Extension to multi-class BCMP networks with class switching.
- Tool support for sensitivity analysis (how performance metrics change with parameter variations).
- Integration with observability platforms for runtime performance monitoring.

The full specification (12 chapters, ∼4,600 lines) is available in the accompanying repository.

References

A Example: CI/CD Pipeline (Excerpt)

```
type PullRequest = {
  id: Int, author: String, title: String,
  files_changed: Int, status: PRStatus, priority: Int
}

channel pr_queue : Chan<PullRequest>
channel build_queue : Chan<PullRequest>
channel review_queue: Chan<BuildResult>
channel rework_queue: Chan<ReviewResult>

resource build_agents : Resource<4>
resource review_capacity: Resource<3>

station BuildServer : build_queue -> review_queue {
  servers: 4
  discipline: fifo
  service_time: LogNormal(log(8m), 0.6)
  compute {
    fn: fn(pr) -> build(pr)
    service_time: LogNormal(log(8m), 0.6)
  }
}

station CodeReview : review_queue -> test_queue {
  servers: 3
  discipline: fifo
  service_time: LogNormal(log(45m), 0.8)
  yield: Bernoulli(0.70)
  rework: { probability: 0.30, target: rework_queue }
  human {
    role: "senior_engineer"
    sla: 4h
    service_time: LogNormal(log(45m), 0.8)
    schedule: { 9h..17h }
  }
}

arrival PRArrivals : {
  channel: pr_queue
  distribution: Exponential(10/d)
  job: { id: 0, author: "dev", title: "feature",
         files_changed: 5, status: Pending, priority: 1 }
}

assert throughput(CICDPipeline) >= 9.5/d
assert cycle_time(CICDPipeline).p95 <= 2d
assert bottleneck(CICDPipeline) == CodeReview
assert deadlock_free(CICDPipeline)
assert littles_law(CICDPipeline)
```