Lightweight, Simulation-Based Software Delivery Process Improvement

Facilitating model-driven engineering and simulations in software delivery using a domain-specific discrete event simulation platform

Nodematic Labs

Abstract

While model-driven engineering and simulation are commonplace for software development, they are rarely applied to the domain of software delivery. Software delivery, the people, processes, and tools involved in bringing software developments to the market, is a core component of the software application lifecycle. Mastering software delivery leads to innovation acceleration, cost reduction, and risk mitigation. The limited use of model-driven engineering and simulation in software delivery could be due to limited frameworks, platforms, and off-the-shelf products to facilitate this engineering work. A lightweight, domain-specific, and easy to use simulator could be pivotal in bringing model-driven engineering and simulation to software delivery. This report investigates the feasibility of combining lightweight client-side technologies and a focus on *time to value* to create the foundation for a software delivery simulation platform.

CONTENTS

I. Background

Enterprises and small businesses alike face challenges in delivering new software capabilities to the market. The software development itself is only one piece of the puzzle. Software delivery, the process of getting new software developments out to the market, can add considerable cost, risk, and time to the firm's software projects. Simulation, for systems analysis and design, may hold great potential as a tool to improve software delivery systems.

Organizations are constrained in their ability to carry out simulation-based process improvement projects. With software providing the backbone for many organization's value proposition, the baseline, "day-to-day" demands on software engineers are high. With a continued skilled labor shortage in software engineering, many organizations are constrained in their ability to scale engineering capacity and meet these day-to-day demands, let alone execute non-business-critical projects [1]. The inability to undertake simulation-based process improvement projects is exacerbated by the lack of off-the-shelf tools to facilitate this simulation. The resource demands are simply too high - leading organizations to skip simulation-driven, model-driven, and data-driven software delivery improvement projects. A purpose-built software delivery simulation platform may bridge the gap - making simulation-driven process improvement a reality for more organizations.

II. Simulator Design

A. Principle Objective

The Software Delivery Simulator is an experiment in optimizing around *time to value*, while using modern, lightweight web technologies. A lower *time to value* means less time and effort required to create valuable simulations and insights - a benefit which would accelerate the adoption of simulation in the software delivery domain. The experimental question is investigated through direct practice - designing, implementing, analyzing, and validating a simulator.

B. Design Tradeoffs

The lens of design tradeoffs can provide unique insights into the philosophy, architecture, and implementation of a simulator. A particularly insightful tradeoff analysis for the simulator is across the domains of "ease of use", "problem domain breadth", and "level of detail". While not mutually exclusive, per se, optimizations around one dimension tend to negatively impact the others. For example, a simulator with no constraints on the problem domain and capabilities for arbitrary levels of simulation detail would be exceptionally challenging to use. In fact, this would essentially be a general-purpose programming language. A custom-made application, built for a specific industry, specific organization, and specific problem, could have exceptional simulation detail and ease of use, but this is only possible by constraining the problem space to a single specific problem. For the Software Delivery Simulator, optimizing around *time to value* suggests a heavy weighting of "ease of use". Of the other two factors, simulation detail is preferred over breadth of problem domain.

C. Key Features

The key features of the simulator arise from the principle objective: optimizing around *time to value*, while using modern, lightweight web technologies. In accordance with this objective, the simulator features:

1) A no-code approach for end user simulator use
2) Alignment with the Discrete Event System Specification (DEVS) formalism
3) Business Process Modeling Notation (BPMN) components and representations
4) Scalable, purely client-side browser-based technologies

1) No-Code: Historically, efforts to simplify the modeling process have been over-optimistic. Attempts to create high-level to low-level model translations, interactive program generators, and natural language interfaces have encountered severe limitations in modeling generality - this machinery is unable to accommodate the unavoidable complexity of real world systems [2]. However, most success has been seen with simulation systems designed for narrow domains [2]. This is precisely the strategy adopted by the Software Delivery Simulator - heavily constrain the problem domain in order to dramatically improve ease of use and simulation project *time to value*.

A no-code approach for software delivery simulation dramatically lowers the barrier to entry for simulator use. A no-code design makes the simulator accessible for engineering managers, project management professionals, and business analysts - people who often have a high stake in software delivery systems, but do not necessarily have a simulation engineering skill set. Templatization, collaboration structures, and sharing systems can further improve the no-code value proposition by enabling even faster simulation projects (via reuse) and by connecting simulations to best practices and common patterns.

2) Discrete Event System Specification: While the choice of modeling formalism most significantly affects the simulator engine, the user experience and simulation representation formats are also impacted. The Discrete Event System Specification (DEVS) is a natural choice of formalism, due to its modularity, hierarchical constructions, rigorous formal definition, and intuitive core abstractions. The modularity and hierarchical constructions in DEVS facilitate the modeling of large, complex sociotechnical systems (e.g., software delivery). Modular atomic models can be defined and reused for the people, processes, and tools in software delivery - such as continuous integration tools, chat systems, and manual approval processes. Systems-of-systems can be broken down into manageable pieces, while maintaining the possibility of emergent behavior in the resulting simulations. The rigorous formal definition imposes a concrete structure to the simulator and models, for predictable behavior and controlled addition of new models and extensions. The model couplings and inter-component messages of DEVS are conceptually similar to the communication across humans and machines in software delivery.

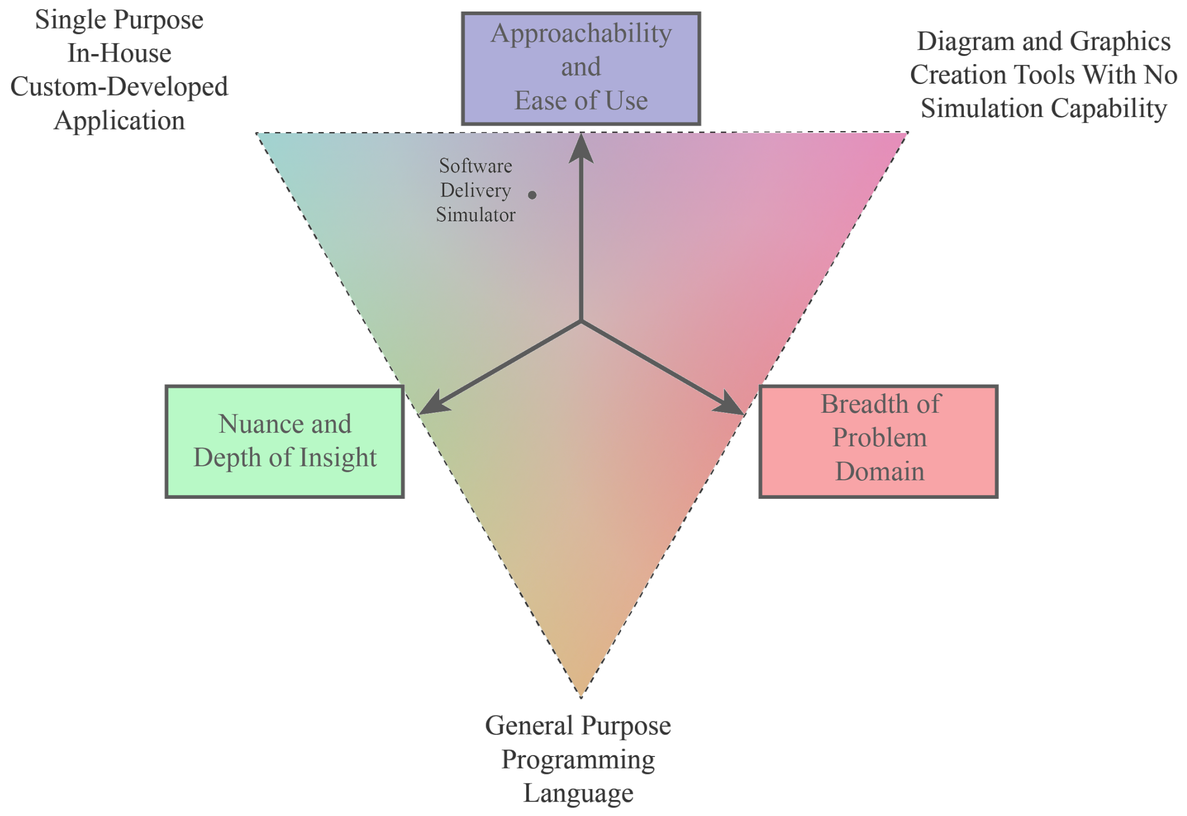

Single Purpose In-House Custom-Developed Application
Approachability and Ease of Use
Diagram and Graphics Creation Tools With No Simulation Capability
Software Delivery Simulator
Nuance and Depth of Insight
Breadth of Problem Domain
General Purpose Programming Language

Fig. 1. Software Delivery Simulator design tradeoffs

3) Business Process Modeling Notation: The DEVS formalism will underpin the simulator operation, in the simulator engine, but the addition of BPMN structures (encapsulated in DEVS models) will make the simulator more intuitive. The target audience for the simulator includes people with no direct software development experience and no simulation engineering experience. For these users, BPMN may offer a thread of familiarity that makes the simulator less imposing.

4) Browser-Based, Client-Side: Modern software applications are increasingly run over the internet, and provided as a service - not just packaged and shared over the internet. This shift has many implications, but most importantly, for simulation engineering is the increased portability, accessibility, and approachability of simulators. Modern web development technologies account for, and abstract away, the nuances of browser implementations. In practice, a web application can be written once and used on nearly any operating system, nearly any browser, and nearly any device. The user is provided with HTML, JavaScript, CSS, and WebAssembly, and the simulation work can begin. No installs, no special compiling, and (almost) no server-side requirements. Users are naturally hesitant about traditional software installs on their machines, particularly in professional settings. Software installs carry long-term resource demands like disk space, can "crowd" a machine (making it more difficult or less enjoyable to operate), and require security due diligence. These concerns are non-existent, or at least mitigated, for a browser-based web application. All else equal, a browser-based application will be more accessible and approachable for new users.

The simulator will be delivered with Cloudflare's cloud-native, "edge computing" technologies. This method of hosting is only possible for client-side simulator web applications, so is relatively unexplored in both academic and professional simulator projects. SMB commercial applications and academic projects, in particular, could benefit from this architecture, based on the alignment of low budgets and low system operation costs. Despite the low costs, the performance of this web application architecture could be exceptional.

In addition to potential cost and performance benefits, a client-centric architecture holds potential for risk reduction. Maintaining and updating large server-centric software systems can be risky and time-consuming. The high risk and resource requirements are driven by system couplings, team inter-dependencies, and system-of-systems complexity. The deployment of a typical enterprise Java application could require deep expertise across networking, security, infrastructure, frontend web technologies, backend web technologies, and databases. By using a fully client-side implementation, on an edge computing, cloud-native platform, people responsible for the application only need to concern themselves with "pushing out the new frontend".

D. Event Relationship Graphs for Atomic Models

While the DEVS formalism provides a concrete structure for the simulation, the atomic model definitions can quickly become complex. The standard algebraic semantics used for DEVS atomic models can hinder conceptualization and rationalization. A graphical modeling language, used to define atomic models, may help atomic model designers, implementers, and users overcome modeling complexity.

Considerable precedent exists for dynamical system representation via modeling languages, and particularly UML. However, UML and its variants lack an efficient and effective way to characterize atomic DEVS models. A combination of UML sequence diagrams and UML state diagrams might be required to characterize model behavior - leading to a disjointed model definition. Event relationship graphs are an efficient and effective alternative - enabling full atomic model characterization in a single graph [3]. While aligned particularly well to the domain of discrete event simulation, event relationship graphs are more broadly able to characterize any dynamic system [3]. Event relationship graphs are Turing complete [4].

The best precedent for event relationship graphs as a modeling language for discrete event simulation might be SIGMA. From the SIGMA website [5]:

> "SIGMA is based on the simple and intuitive Event Relationship Graph (sometimes called an ERG or Event Graph) approach to simulation modeling. The SIGMA project began as an effort to implement the notion of Event Relationship Graphs on personal computers and has evolved into a powerful and practical method for simulation modeling. SIGMA, the Simulation Graphical Modeling and Analysis system, is an integrated, interactive approach to building, testing, animating, and experimenting with discrete event simulations, while they are running. SIGMA is specifically designed to make the fundamentals of simulation modeling and analysis easy."

Event scheduling, state variables, and state transitions within atomic models can be handled with standard event relationship graph notation and construction. However, the Discrete Event System Specification (DEVS) message exchanges among atomic models require special attention. Sending and receiving messages is not native to the event relationship graph approach, but can be accommodated with relative ease. For the Software Delivery Simulator, we consider receiving a message to be an event, within the event relationship graph framework. Sending a message is treated as a state change operation.

The event relationship graph for the generator atomic model is shown in Figure 3. This atomic model is a good example of event relationship graph concepts, because it is relatively complex among the set of Software Delivery Simulator atomic models, and contains a wide variety of elements.

The event relationship graph characterizes the dynamic system that is the Generator atomic model - a model with stochastic generation of messages. The generator immediately begins generating these messages, upon model initialization. Any message received on the deactivation port for the generator will cause the generator to passivate and cease all message generation. Subsequent activation by a message on the activation port will resume generator operations.

The "Run" event and associated state variable definitions set the initial conditions and parameterization of the model. The two P_{in} ports are dedicated to activation and deactivation. The single P_{out} port is for sending generated jobs. The interdeparture time of messages is characterized by an exponential distribution, of rate λ. The generated job number is initialized to 0, and incremented as new jobs are generated.

The "Begin Generation" event is where a random variate from the exponential distribution is generated. Furthermore, the event self-schedules with an arrow directed at itself, with a delay of t_i (the value of the exponential random variate). After the delay of t_i, should no other event change the scheduling, the "Send Job" event is triggered, along with a new "Begin Generation". The "Send Job" increments the job number and sends a message on the output port, while the "Begin Generation" maintains the loop of generation events.

The two Message events are triggered upon receiving a message at either the activation or deactivation port. In the former case, the "Begin Generation" event is rescheduled at no delay, if the generator is currently passive (has been deactivated). In the latter case, job generation and message sending are canceled. The event cancellations from the deactivation event create a generator state where no events are scheduled. Unless an activation message is received, the generator will remain passive and do nothing through perpetuity.

E. Hierarchical Simulation Constructions

The ability to hierarchically compose models provides immeasurable value for modelers. Human minds are limited in the ability to conceptualize and work with large interconnected systems. Composing systems into systems of (smaller) systems makes the simulation work far more approachable to simulation engineers. In the application architecture, it's important to make this realization, that hierarchical constructs are useful almost exclusively on the "human side" of the simulation construction. The computer is not similarly challenged with networks of large interconnected models. In fact, logical structures that have no impact on simulation execution have the potential to deteriorate system performance.

Given the componentization of the simulation engine (WebAssembly) and frontend (traditional web application technologies), hierarchical constructs are best addressed as a pure frontend consideration. Users can leverage hierarchical

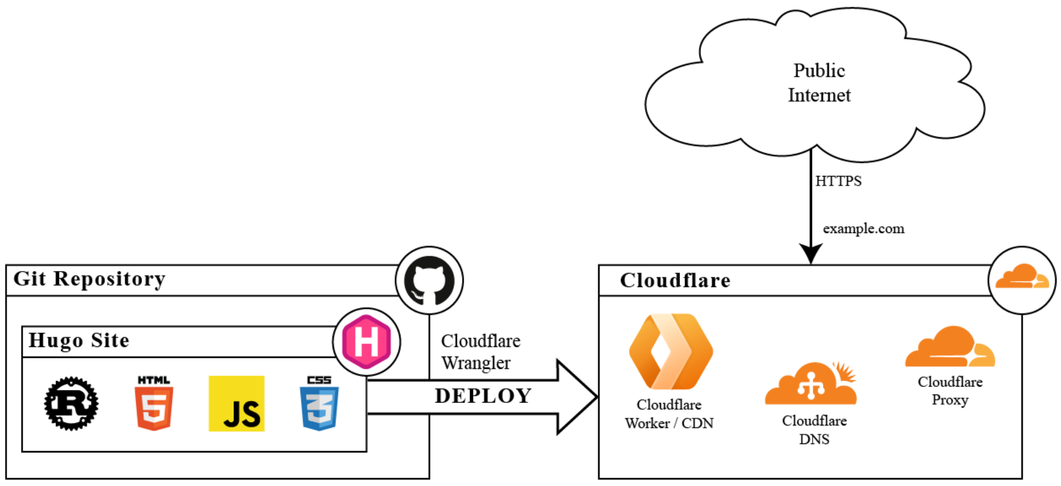


Fig. 2. Web application infrastructure, deployment, and tooling systems

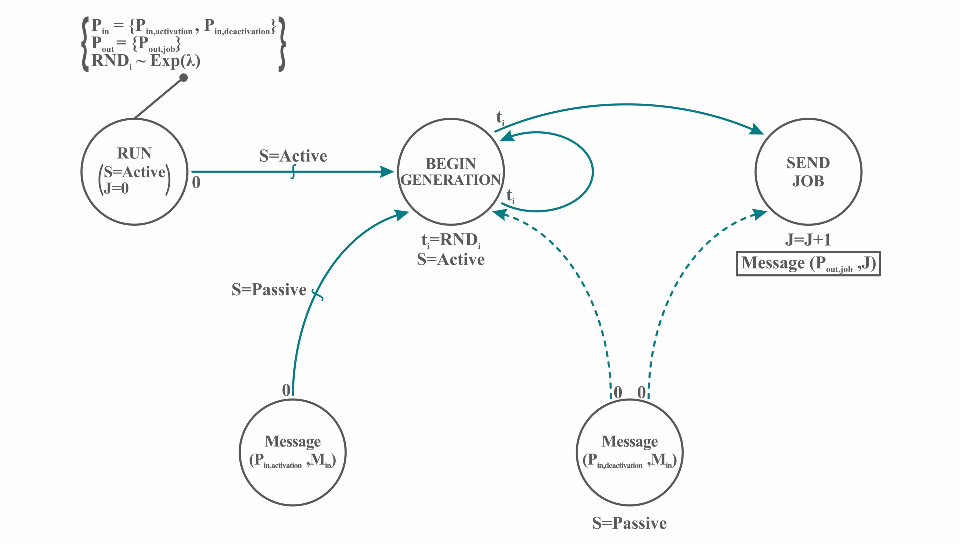

$P_{in} = \{P_{in,activation}, P_{in,deactivation}\}$
$P_{out} = \{P_{out,job}\}$
$RND_i \sim Exp(\lambda)$
RUN (S=Active J=0)
S=Active
0
BEGIN GENERATION
t_i
t_i=RND_i
S=Active
SEND JOB
J=J+1
Message ($P_{out,job}$,J)
S=Passive
0
Message ($P_{in,activation}$,M_{in})
0 0
Message ($P_{in,deactivation}$,M_{in})
S=Passive

Fig. 3. Generator event relationship graph

constructions, for ease of modeling, but those hierarchies are flattened before execution.

III. Simulator Implementation

A. Technology Choices

The technology choices support the aforementioned design objectives. The resulting implementation exhibits the following:

- Written in Rust and executed in the browser as WebAssembly.
- Uses a YAML configuration format to define model configurations, initial conditions, component connections, and system state during execution.
- Implements atomic and coupled models to simulate code repository developments, software builds, artifact scans, systems tests, deployments, notifications, and manual software delivery tasks.
- Leverages hierarchical model constructions, with coupled components.
- Includes a configurable implementation of key BPMN 2.0 gateways “Exclusive” and “Parallel”.
- Accessible over a public URL, and featuring delivery with CloudFlare Workers - CloudFlare’s commercial cloud-native, edge computing technology.
- Leverages JSON string inter-component messages.

While future implementations will include a graphical “drag-and-drop” frontend interface, the initial implementation instead focuses on the simulator core - complementing that on the frontend with just a simple text editor web page:

- Input box (yaml configuration of initial simulation state)

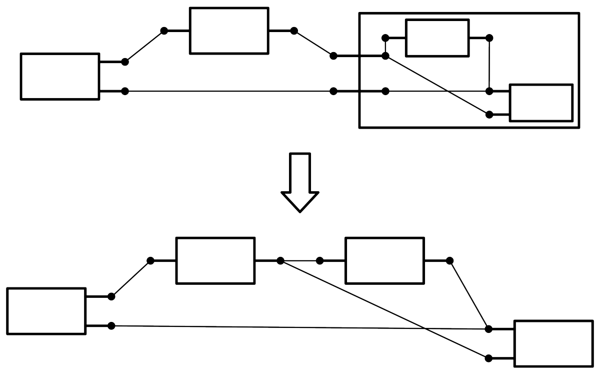


Fig. 4. Flattening hierarchical model constructions before execution

- Output box (yaml representation of current simulation state)
- A set of controls for stepping through the simulation and manual message injection

B. WebAssembly

The simulator will take advantage of modern WebAssembly technologies and tooling. WebAssembly's objective is to provide safe, fast, portable low-level code on the web [6]. As an abstraction on top of modern hardware, it is not tied to any specific language, hardware, or platform. While WebAssembly is not the first attempt at low-level code on the web, it is the first truly portable solution for fast low-level code, is an official W3C recommendation, and is adopted by all the major browsers [6], [7].

WebAssembly dethrones JavaScript as the only natively-supported programming language on the web (a position attained through "historical accident"), but it does not replace JavaScript [6]. As a complement to JavaScript, the addition of WebAssembly gives application developers more capabilities and flexibility. Furthermore, to arrive at the final JavaScript and WebAssembly running in the browser, a wide variety of development technologies and programming languages can be used. Web applications can be developed with languages, frameworks, and tooling that compile to WebAssembly - expanding on the historical limitation of JavaScript and languages that transpile to JavaScript. The expanded solution space for development technologies increases the overlap between the problem spaces and solutions spaces.

For simulation applications specifically, WebAssembly provides not only the simulation execution benefits of safe, fast, portable low-level code, but also the developer experience benefits of using programming languages and technologies that are well suited to simulation development, but are historically poorly suited to web development. The performance of WebAssembly improves the simulator *time to value*, not just in experimental run duration, but also in terms of simulation setup time. Setting up a simulation may require many cycles of development, execution for verification/validation, and debugging - a performant simulator will accelerate these cycles.

C. Rust Programming Language

The value of the Rust programming language is becoming apparent, and it's poised to possibly become the de facto standard for systems and lower-level programming (a replacement to C and C++). At Open Source 101 at Home 2020, Ryan Levick (from Microsoft), claims that Rust is the industry's best chance at safe systems programming [8]. 70% of CVEs at Microsoft are attributable to memory safety issues, and the associated cost of these memory safety issues is conservatively in the billions USD [8]. Assessments like these suggest an exceptionally high value in modern C/C++ replacements, like Rust. With simulators being commonly written in C and C++, Rust is inherently an interesting candidate for simulation development. Rust is known to be used at Microsoft, Facebook, Amazon, Google, DropBox, Intel, ARM, CloudFlare, and Mozilla [9].

Object-oriented programming languages are a recommended, natural choice for DEVS software development [10]. However, there is no consensus on what specifically defines an object-oriented programming language. Rust does meet the definition, as defined in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, "Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations," [11]. However, the benefits of inheritence often underpin recommendations around object-oriented programming for DEVS [10]. Rust does not support inheritance. *The Rust Programming Language* suggests [12]:

> "Inheritance has recently fallen out of favor as a programming design solution in many programming languages because it's often at risk of sharing more code than necessary. Subclasses shouldn't always share all characteristics of their parent class but will do so with inheritance. This can make a program's design less flexible. It also introduces the possibility of calling methods on subclasses that don't make sense or that cause errors because the methods don't apply to the subclass. In addition, some languages will only allow a subclass to inherit from one class, further restricting the flexibility of a program's design."

Going one last step deeper, the suggestion for inheritance in DEVS programming may ultimately boil down to the benefits of polymorphism, encapsulation, abstraction and code reuse [10]. The Rust programming language includes all of these features. A comparison of Rust with traditional, inheritance-heavy object-oriented programming is presented in Table I.

D. Components and Modules

A good simulator will not only abide by good software architecture principles, but also reflect effective design patterns for simulators specifically. The Rust code, which makes up the simulator, is broken into many modules and submodules. Encapsulation is leveraged so that module interfaces are controlled. The frontend requires much less code and complexity,

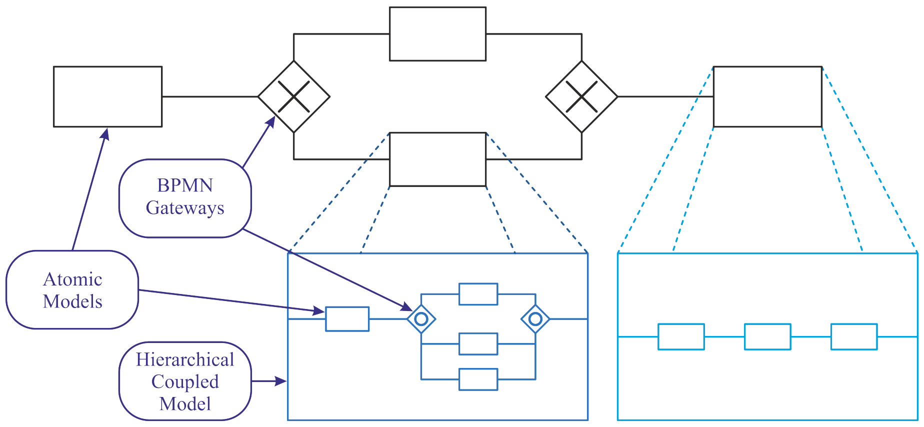


Fig. 5. Hierarchical model constructions and BPMN

TABLE I
A COMPARISON OF RUST FEATURES TO OBJECT ORIENTED PROGRAMMING FEATURES

Feature	**Object Oriented Programming**	**Rust**
Code reuse	Inheritence	Default trait method implementations
Polymorphism	Subclasses (inheritence)	Trait objects
Encapsulation	Yes	Yes
Abstraction	Yes	Yes

compared to the Rust simulator core. If the frontend were to grow in size and complexity in future work, well-established patterns for JavaScript code quality and organization can be leveraged. The most important parts of the codebase are the simulator and models modules, and the associated submodules. The simulator module handles executing the simulation, and the network of associated models. The models module contains the atomic models (each its own submodule) which define the behavior of that specific model. The design pattern optimizes around componentization and modularity.

Establishing the scope of operations and responsibilities for a simulator and it's models is a delicate balancing act. Essentially, repetitive logic and structures can and should be extracted from models, and brought into the simulator. Meanwhile, the models should be relatively independent, modular, componentized, and autonomous - with enough self-contained logic and structures to describe the associated real-world phenomenon. The Software Delivery Simulator balances the scope of these two logical systems, as shown in Figure 8.

E. Creating and Extending Models

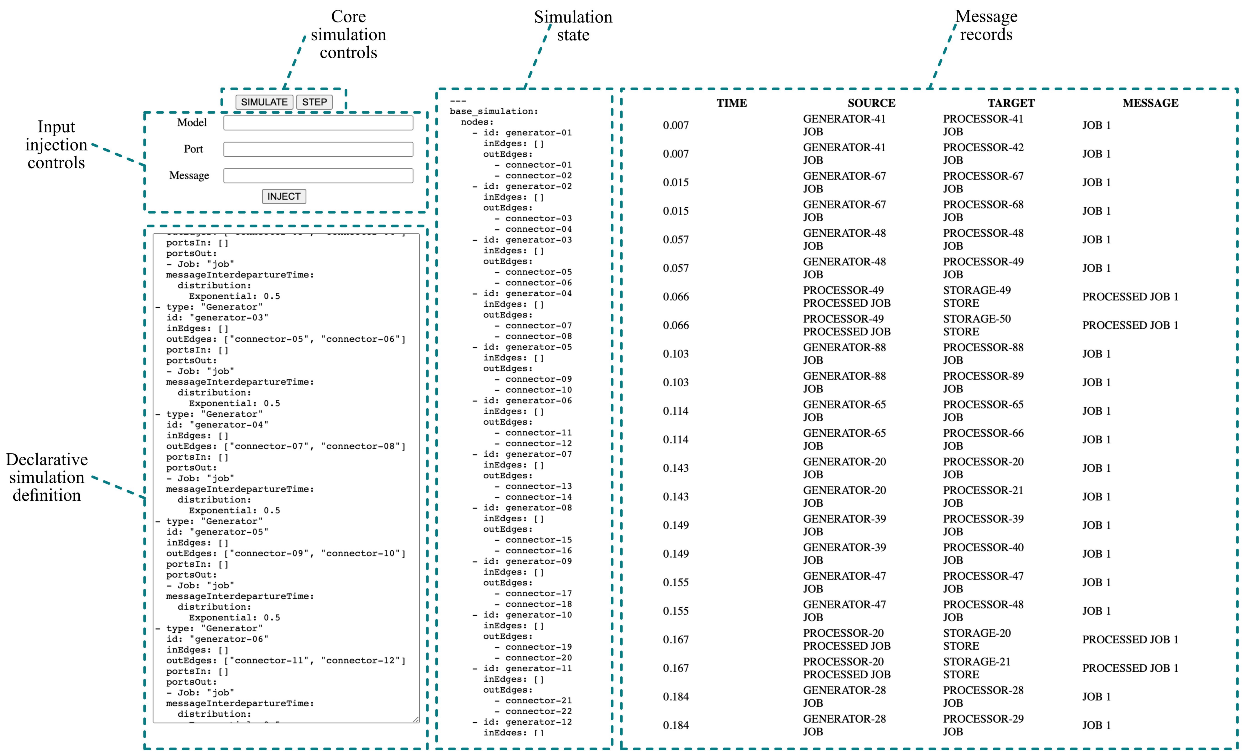
While a good simulator can be reused across a wide variety of applications, models may not be as portable across simulation problems. New use cases will involve new phenomenon that can only be captured by new models. As such, the amount of boilerplate code and "plumbing" to create a new model has major implications for extensibility. Given the *time to value* objective of the Software Delivery Simulator, extensibility is a relatively low priority, but it is nonetheless considered. In the Software Delivery Simulator, two short functions are required for essential model plumbing, while another four functions are used to define the specific dynamics and behavior of the model.

1) Model Plumbing: Upcasting and ID access are required for the model to properly operate within a larger simulation. These functions are fairly generic, "boilerplate" model code.

2) Events External: In alignment with the DEVS formalism, this function triggers model behavior based on messages received by the model [13]. The model behavior may depend on the port that the message is received on and/or the message contents. Messages are strings, but more complicated behavior can be modeled by passing JSON strings, and parsing these upon message arrival. The events external function can itself produce output messages.

3) Events Internal: The model events list contains a list of scheduled events that the model will execute after set periods of time have elapsed. These events can be cancelled or changed - most commonly due to an "events external" execution. If no such disruption occurs, and the time until the next event in the list elapses, then the events internal function will trigger the model behavior. While the events internal execution does not occur due to message arrival, the execution can generate output messages, for transmission to other models.

4) Time Advance: A model's event list contains the scheduled list of events, should no disruptions occur. The time advance function reduces the time until the event, for all the events in the list. This time advance may bring the time until the next event to zero - in which case, the associated events will fire. It is possible for the time advance function to do



```
  portsIn: []
  portsOut:
  - Job: "job"
  messageInterdepartureTime:
    distribution:
      Exponential: 0.5
- type: "Generator"
  id: "generator-03"
  inEdges: []
  outEdges: ["connector-05", "connector-06"]
  portsIn: []
  portsOut:
  - Job: "job"
  messageInterdepartureTime:
    distribution:
      Exponential: 0.5
- type: "Generator"
  id: "generator-04"
  inEdges: []
  outEdges: ["connector-07", "connector-08"]
  portsIn: []
  portsOut:
  - Job: "job"
  messageInterdepartureTime:
    distribution:
      Exponential: 0.5
- type: "Generator"
  id: "generator-05"
  inEdges: []
  outEdges: ["connector-09", "connector-10"]
  portsIn: []
  portsOut:
  - Job: "job"
  messageInterdepartureTime:
    distribution:
      Exponential: 0.5
- type: "Generator"
  id: "generator-06"
  inEdges: []
  outEdges: ["connector-11", "connector-12"]
  portsIn: []
  portsOut:
  - Job: "job"
  messageInterdepartureTime:
    distribution:
```

```
---
base_simulation:
  nodes:
    - id: generator-01
      inEdges: []
      outEdges:
        - connector-01
        - connector-02
    - id: generator-02
      inEdges: []
      outEdges:
        - connector-03
        - connector-04
    - id: generator-03
      inEdges: []
      outEdges:
        - connector-05
        - connector-06
    - id: generator-04
      inEdges: []
      outEdges:
        - connector-07
        - connector-08
    - id: generator-05
      inEdges: []
      outEdges:
        - connector-09
        - connector-10
    - id: generator-06
      inEdges: []
      outEdges:
        - connector-11
        - connector-12
    - id: generator-07
      inEdges: []
      outEdges:
        - connector-13
        - connector-14
    - id: generator-08
      inEdges: []
      outEdges:
        - connector-15
        - connector-16
    - id: generator-09
      inEdges: []
      outEdges:
        - connector-17
        - connector-18
    - id: generator-10
      inEdges: []
      outEdges:
        - connector-19
        - connector-20
    - id: generator-11
      inEdges: []
      outEdges:
        - connector-21
        - connector-22
    - id: generator-12
      inEdges: []
```

TIME	SOURCE	TARGET	MESSAGE
0.007	GENERATOR-41 JOB	PROCESSOR-41 JOB	JOB 1
0.007	GENERATOR-41 JOB	PROCESSOR-42 JOB	JOB 1
0.015	GENERATOR-67 JOB	PROCESSOR-67 JOB	JOB 1
0.015	GENERATOR-67 JOB	PROCESSOR-68 JOB	JOB 1
0.057	GENERATOR-48 JOB	PROCESSOR-48 JOB	JOB 1
0.057	GENERATOR-48 JOB	PROCESSOR-49 JOB	JOB 1
0.066	PROCESSOR-49 PROCESSED JOB	STORAGE-49 STORE	PROCESSED JOB 1
0.066	PROCESSOR-49 PROCESSED JOB	STORAGE-50 STORE	PROCESSED JOB 1
0.103	GENERATOR-88 JOB	PROCESSOR-88 JOB	JOB 1
0.103	GENERATOR-88 JOB	PROCESSOR-89 JOB	JOB 1
0.114	GENERATOR-65 JOB	PROCESSOR-65 JOB	JOB 1
0.114	GENERATOR-65 JOB	PROCESSOR-66 JOB	JOB 1
0.143	GENERATOR-20 JOB	PROCESSOR-20 JOB	JOB 1
0.143	GENERATOR-20 JOB	PROCESSOR-21 JOB	JOB 1
0.149	GENERATOR-39 JOB	PROCESSOR-39 JOB	JOB 1
0.149	GENERATOR-39 JOB	PROCESSOR-40 JOB	JOB 1
0.155	GENERATOR-47 JOB	PROCESSOR-47 JOB	JOB 1
0.155	GENERATOR-47 JOB	PROCESSOR-48 JOB	JOB 1
0.167	PROCESSOR-20 PROCESSED JOB	STORAGE-20 STORE	PROCESSED JOB 1
0.167	PROCESSOR-20 PROCESSED JOB	STORAGE-21 STORE	PROCESSED JOB 1
0.184	GENERATOR-28 JOB	PROCESSOR-28 JOB	JOB 1
0.184	GENERATOR-28 JOB	PROCESSOR-29 JOB	JOB 1

Fig. 6. Running a large simulation in the GUI

more than just reschedule the events list. For example, non-stationary model behavior would be driven by the time advance function. In the Software Delivery Simulator, no such non-stationary model behavior exists.

5) Until Next Event: The "until next event" function simply checks the next scheduled event(s) from the event list, and reports the elapsed time required to reach those scheduled events. A simulation with many models will use this until next event function, across all models in the simulation, to understand the global time until the next event.

IV. Verifications and Validations

Simulator and model verifications and validations is a research field in it's own right. Challenges, like how to actually know the expected result from a test simulation, require unique verification and validation methods, compared to other software applications.

For the Software Delivery Simulator, there are two core levels of testing. The first is low-level verifications - checking for program correctness with unit-like testing, as is common with other software applications. The second level of testing is test simulations, which serve some level of verification, as well as validation.

The test simulations are complicated enough to verify complicated simulator and atomic model behavior, but are typically simple enough to have closed-form analytical results. For example, Markovian processor chains can be analyzed through queueing theory, and the subsequent results can be used within the test oracle. For effective test simulations, the simulator leverages:

- Seeded random number generation, for deterministic tests.
- Confidence intervals, instead of point estimates, for test simulation outputs.
- Varied strategies for initial conditions specification, including reducing initialization bias with time series processing techniques.
- Queuing theory utilization, for determination of some expected results.

V. Simulator Analysis

With functional capabilities of the simulator verified by the two levels of automated testing, a foundation is established for the analysis of non-functional attributes. Non-functional attributes of particular interest are performance and usability. The focus on these non-functional attributes arises from the project focus on a lightweight, low time-to-value simulator. Performance analysis ensures that the simulator shows sufficient performance, despite the use of only browser-based, client-side resources. The usability analysis sheds light into how intuitive the application is, throughout the simulation lifecycle, so that value can be rapidly realized in a simulation project.

A. *Performance Analysis Setup*

For this performance testing, the application is loaded, a simulation initialized, and the simulation is executed. The

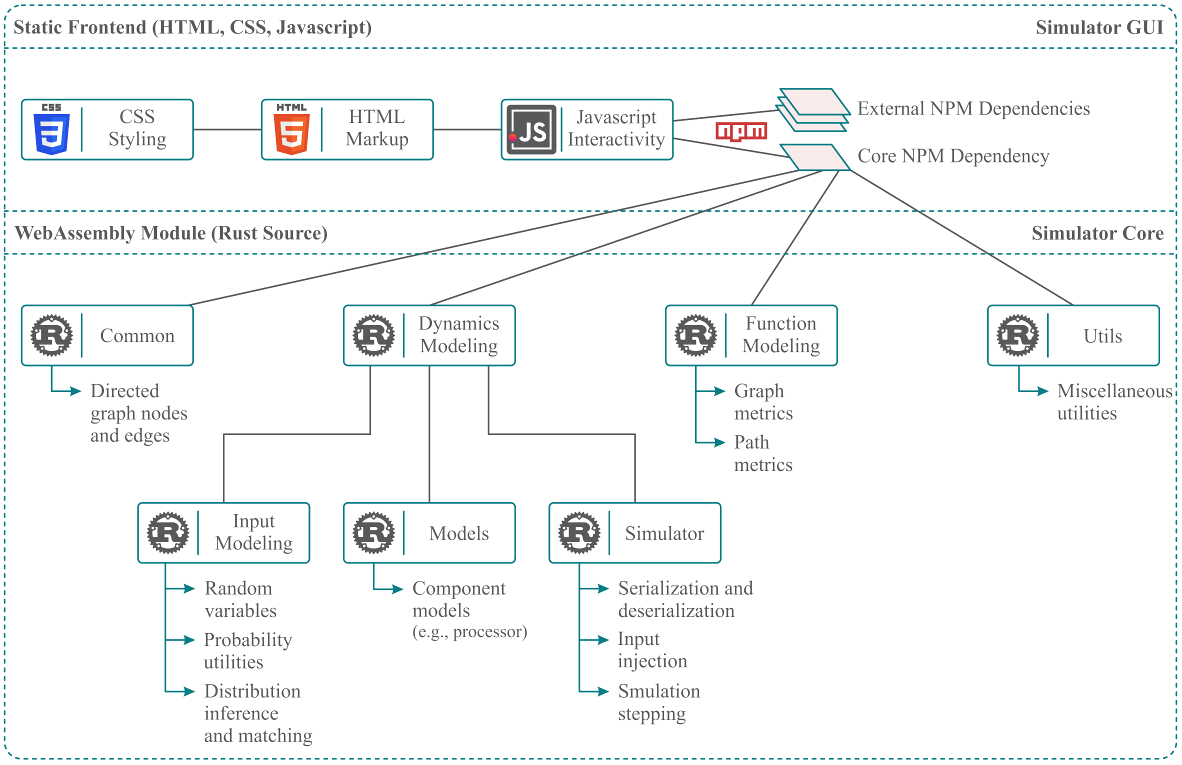


Fig. 7. Layers and modules of the simulator

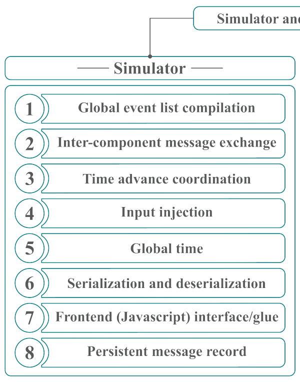

Simulator and Model Scoping
Simulator
1 Global event list compilation
2 Inter-component message exchange
3 Time advance coordination
4 Input injection
5 Global time
6 Serialization and deserialization
7 Frontend (Javascript) interface/glue
8 Persistent message record
Model
1 Model event list compilation
2 Model state variables and associated operations
3 Model event execution
4 Receiving message logic
5 Sending message logic
6 Model time advance

Fig. 8. Simulator and model scoping

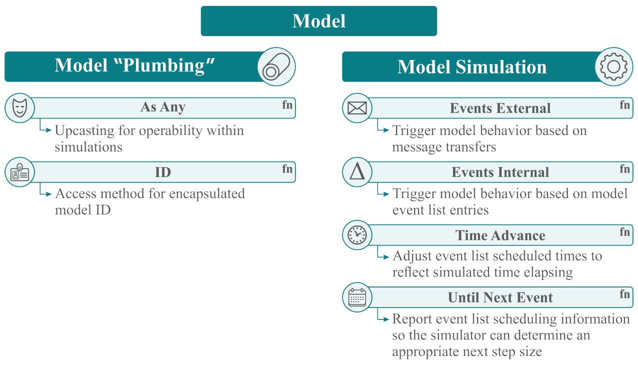

Model
Model "Plumbing"
As Any
fn
Upcasting for operability within simulations
ID
fn
Access method for encapsulated model ID
Model Simulation
Events External
fn
Trigger model behavior based on message transfers
Events Internal
fn
Trigger model behavior based on model event list entries
Time Advance
fn
Adjust event list scheduled times to reflect simulated time elapsing
Until Next Event
fn
Report event list scheduling information so the simulator can determine an appropriate next step size

Fig. 9. Model functions

example simulation is an extension, or scaling, of a standard generator-processor-storage simulation. All timing is stochastic, and chosen to follow an exponential distribution. Given the stochastic nature of the models, they may behave differently. However, the models in each category (generator, processor, and storage) have an identical model definition and configuration. This setup for simulation testing does not analyze the performance of the simulator in a wide variety of scenarios and projects. Rather, the analysis provides a single lens of insight, by using a specific, highly-structured simulation setup and execution. The performance analysis setup is driven by order-of-magnitude intuitions, rather than concrete frameworks or heuristics. The design can be summarized as *hundreds of models, hundreds of connections, and a handful of simulation steps per second.*

Simulation setup:

- 100 generator models
- 200 generator-to-processor connectors (1:2 connection)
- 100 processor models
- 200 processor-to-storage connectors (1:2 connection)
- 100 storage models
- 300 total models
- 400 total model connectors

This simulation is then executed:

- 10 simulation steps per second

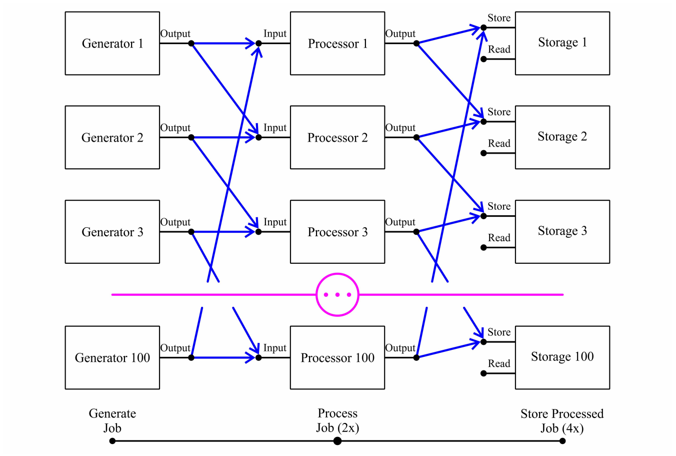
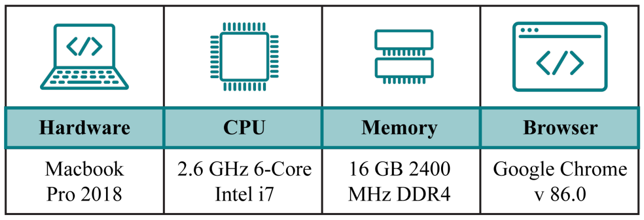


Fig. 10. Load testing configuration of generator, processor, and storage models



Hardware	CPU	Memory	Browser
Macbook Pro 2018	2.6 GHz 6-Core Intel i7	16 GB 2400 MHz DDR4	Google Chrome v 86.0

Fig. 11. Performance testing configuration

- 100 seconds of simulation stepping
- 1000 simulation steps

As a result of the models and couplings, each generated job will be processed twice, and accepted by a storage model four times. The simulation involves a relatively large number of models, connections, and message exchanges, for the problem domain of software delivery simulation.

B. Initialization Performance

A promising possibility for the Software Delivery Simulator, and other simulators with a similar architecture, is an unprecedented cost-to-performance ratio. A simulator application, deployed with “edge technologies” is particularly promising in its initialization performance.

Google PageSpeed Insights are a standard service for evaluating web application initialization. Testing the application with this service provides insights on one key dimension of performance. This performance testing tool has a reputation for being rigorous and very demanding, which can help to differentiate among the high tiers of application performance. For example, the https://gmail.com login page scores a 22 out of 100 for mobile and 81 out of 100 for desktop. The simulator scores a 98 out of 100 for mobile and a 100 out of 100 for desktop.

By combining edge technologies, no images/videos, and a light frontend it’s no surprise that the performance scoring is exceptional. An implementation with additional branding, “getting started” content, and interface styling would see a lower score. However, the tested application configuration shows that a WebAssembly-based simulation engine does not inherently carry a high initial load cost.

In addition to the page/application initialization, one must consider the simulation initialization. This initialization, triggered by the “Simulate” button in the GUI, prepares the simulation for execution. Models are instantiated, hydrated, and connected to a core simulation engine. This initialization consumes the main browser thread, so the interface is inter-operable during this initialization. Initialization is noticeable, but not so long as to be problematic - taking a total of 262 ms. Taking this initialization off the main thread would be a relatively simple operation, but is outside the scope of the initial implementation.

TABLE II
INITIALIZATION TIMING

Initialization Component	Time
Initial configuration yaml deserialization	46 ms
Simulation initial state yaml serialization	43 ms
Simulation initialization	50 ms
Layout (GUI) updates	121 ms

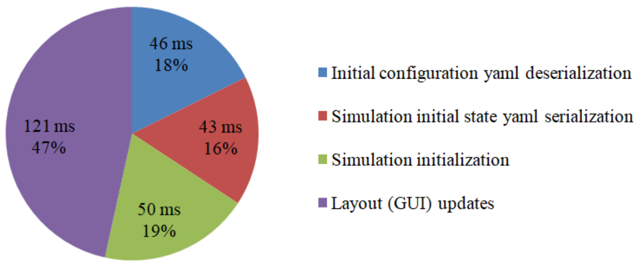


Fig. 12. Breakdown of initialization time

As shown in Table II, the majority of the initialization time is tied up in tasks related to the user interface - yaml serialization, yaml deserialization, and GUI layout updates. Only about 19% of the initialization time is related to the actual core simulation initialization.

Heap memory is allocated for the simulation object during initialization. The performance testing suggests an increase of 1.2 MB (from 4.1 MB to 5.3 MB) in initialization. 1.2 MB of heap memory is unlikely to overwhelm a browser or client machine, so it’s safe to assume that users will be able to reliably initialize simulations of this scale.

C. Steady State Performance

The second key dimension of application performance is steady-state execution. Running the specified configuration of 10 steps per second, for 100 seconds, results in no obvious problems in the GUI (such as freezing or crashing). However, the 100 ms step execution target is not actually realized. At the beginning of the run, steps take about 122 ms. At the end of the run, steps take about 136 ms.

As with the initialization, the vast majority of the time is attributed to user interface work. Near the middle of the run, a typical step takes 0.13 ms to run the simulation step, 13 ms for simulation state yaml serialization, and 120 ms for the layout (GUI) updates. The actual simulation execution takes two orders of magnitude less time than updating the GUI - which is just appending the new message(s) to the message record table and updating the simulation state yaml.

TABLE III
SIMULATION STEP TIMING

Simulation Step Component	Time
Core simulation stepping	0.13 ms
Simulation state yaml serialization	13 ms
Layout (GUI) updates	120 ms

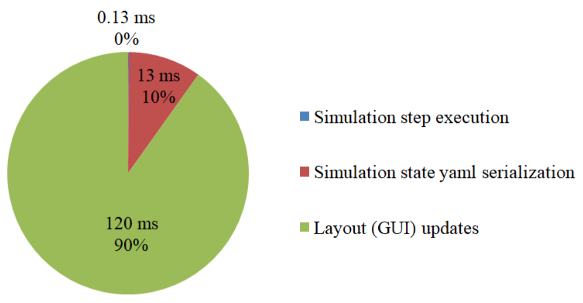


Fig. 13. Breakdown of simulation step timing

The steady-state memory use is no cause for concern. The heap memory fluctuates between 3.1 MB and 4.3 MB. History is not persistent within the simulation core. For example, messages are not recorded after they are generated and subsequently consumed. Message records are, however, maintained in the GUI, within a table of message records. As such, we could expect the heap memory usage to grow over time. However, in practice, an exceptionally long simulation run would be required to hit considerable memory usage.

D. Usability

While a more subjective measurement than performance, usability is critical for understanding the *time to value* for the Software Delivery Simulator. Usability can be conceptually broken down into initial access, simulation setup, simulation execution, and analysis of results.

1) Initial Access: Providing a great user experience during initial simulator access is critical. Users only need to point their browser of choice, on their device of choice, to the URL for the simulator. There is no provisioning, builds, installations, or setup work. Compared to native desktop installs or command-line programs ran on platformized compute environments, it's an order of magnitude improvement to usability.

2) Simulation Setup: Browsers are perhaps the most common interface for interacting with software applications today. Furthermore, graphical user interfaces offer a common and intuitive form of user interaction. While the simulator currently uses a text editor interaction format, a natural extension of the application would be a drag-and-drop interface. Converting from a no-code text editor interface to a no-code drag-and-drop interface would improve the usability, from good to great.

3) Simulation Execution: Dedicated buttons and inputs provide a simple, yet effective, mechanism for simulation execution and control. Like the DEVS Suite, this interface can be a small panel within the larger graphical user interface. The Software Delivery Simulator features simulation setup, simulation stepping, and input injection controls.

4) Analysis of Results: The Software Delivery Simulator, in it's current incarnation, gets low marks for analysis of simulation results. Expansion with plots, tables, animations, data exports, and visualizations would improve the usability of the simulator. However, these features were out of scope for the initial application implementation.

VI. CONCLUSIONS AND FUTURE WORK

By building the Software Delivery Simulator with client-side, modern web application technologies, this project provides evidence for the feasibility of shifting simulator applications to the web, with serverless edge technologies. The simulator is architected for low-cost and high performance, and that design objective was realized in practice. The no-code simulator design, ease of simulator access/use, and prebuilt, domain-tailored models provide a solid foundation for quick value generation in simulation projects.

From the perspective of a full Software Delivery Simulator *platform*, considerable risks and uncertainties remain. This project did not address all the core components of a simulation platform - model building, model debugging, animation, interactive running of models, input data analyzers, and output analyzers [2]. It's possible that by extending the Software Delivery Simulator foundation to a full simulation platform, insurmountable or unnecessarily challenging hurdles will be encountered.

It's clear that there exist some "low-hanging fruit" opportunities for improving the Software Delivery Simulator foundation. For example, optimizations around the user interface have the potential for dramatic improvements. This is particularly true for the steady-state performance, which could see orders of magnitude improvement.

Agent-based modeling is another interesting possibility for further development. It is especially well-adapted to problems like civil violence, infectious disease modeling, and traffic modeling, but it's early in terms of industrial and commercial simulation adoption [2]. The limited process modeling perspective and relatively low adoption in business process modeling reduce the fitness of agent-based modeling for the Software Delivery Simulator, but it remains an interesting formalism for future comparison and investigation.

Some topics considered briefly in the project, such as the use of event relationship graphs to define atomic models, or the relative impacts of user interface work and "core simulation" execution on performance, could be further studied independently. Many of these possible branches of additional investigation are unrelated to the application area of software delivery - suggesting possible wide applicability of the researched phenomenon, methodologies, and patterns.

References

[1] Bureau of Labor Statistics, "Occupational outlook handbook - software developers." [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

[2] J. Banks, J. C. II, B. Nelson, and D. Nicol, *Discrete-event system simulation*, fifth edition ed. Prentice Hall, 2009.

[3] L. Schruben, "Simulation modeling with event graphs," 1983.

[4] E. L. Savage, L. W. Schruben, and E. Yücesan, "On the generality of event-graph models," *INFORMS Journal on Computing*, vol. 17, p. 3, 2005.

[5] Bio-G, SIGMA, and Custom Simulations, "About - sigma." [Online]. Available: http://sigmawiki.com/sigma/index.php?title=About

[6] D. L. Schuff, A. Z. Mozilla, J. Bastien, A. Haas, A. Rossberg, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with webassembly," 2017.

[7] "World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation." [Online]. Available: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[8] R. Levick, "Rust at microsoft : Open source 101," 5 2020. [Online]. Available: https://opensource101.com/sessions/rust-at-microsoft/

[9] J. Jackson, "Microsoft: Rust is the industry's 'best chance' at safe systems programming," 2020. [Online]. Available: https://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming

[10] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Academic Press, 8 2018.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995.

[12] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[13] B. P. Zeigler and H. S. Sarjoughian, "Introduction to devs modeling and simulation with java: Developing component-based simulation models," 2005. [Online]. Available: https://acisms.asu.edu

Appendix A
Model Event Relationship Graphs

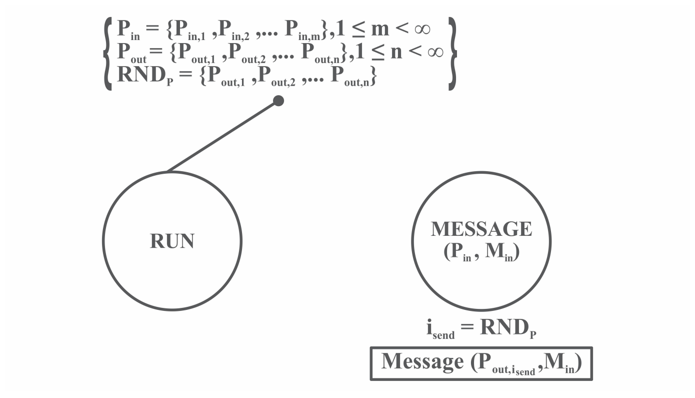

$P_{in} = \{P_{in,1}, P_{in,2}, \dots P_{in,m}\}, 1 \le m < \infty$
$P_{out} = \{P_{out,1}, P_{out,2}, \dots P_{out,n}\}, 1 \le n < \infty$
$RND_P = \{P_{out,1}, P_{out,2}, \dots P_{out,n}\}$
RUN
MESSAGE (P_{in}, M_{in})
$i_{send} = RND_P$
Message $(P_{out,i_{send}}, M_{in})$

Fig. 14. Event relationship graph for the exclusive gateway model

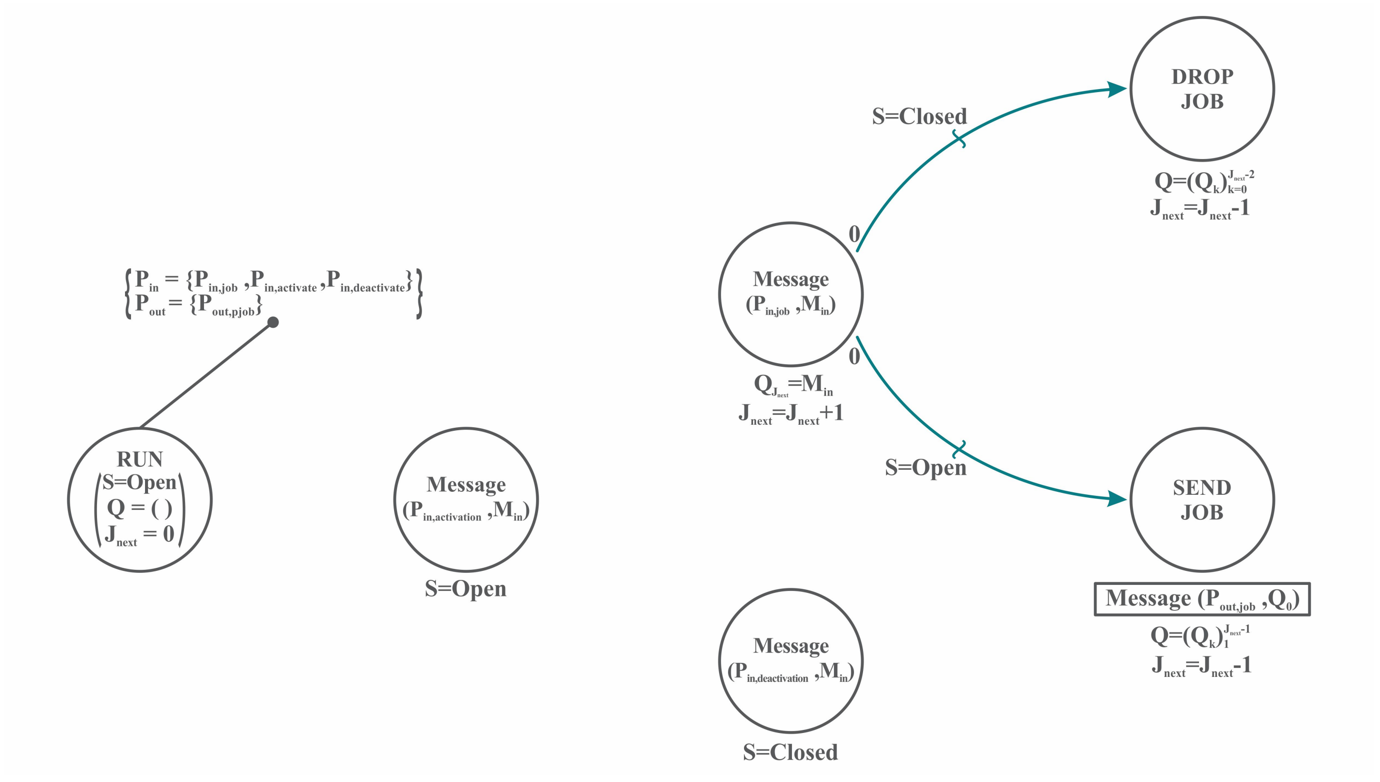
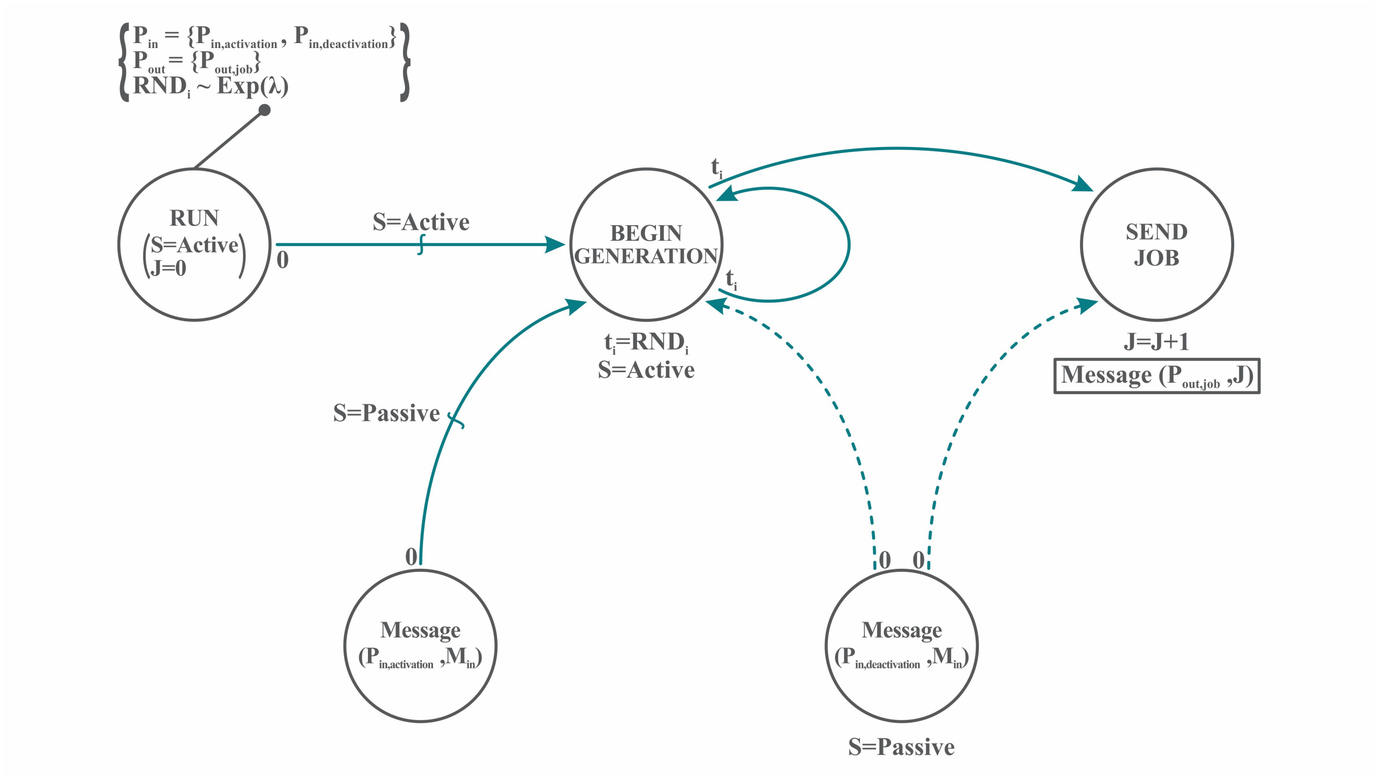
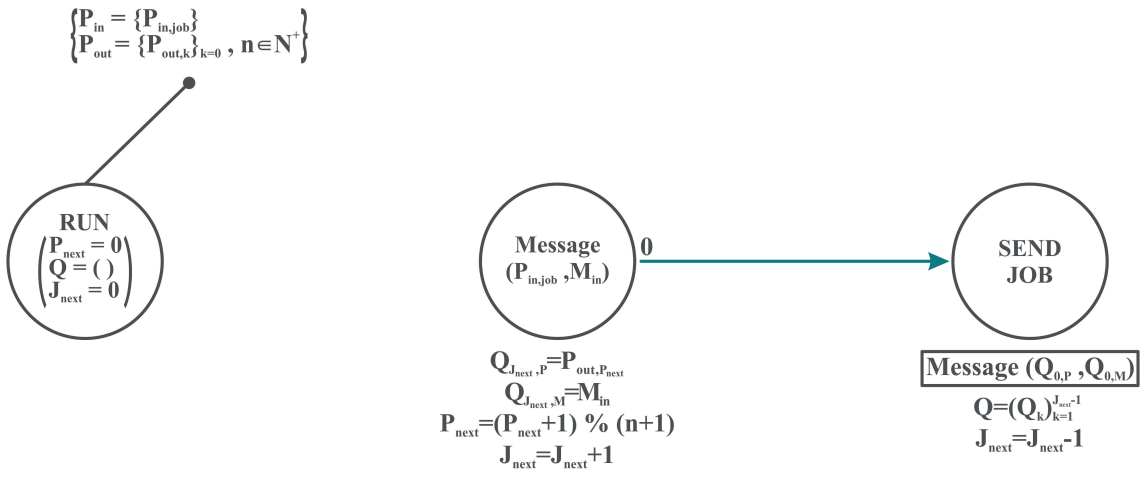

$P_{in} = \{P_{in,job}, P_{in,activate}, P_{in,deactivate}\}$
$P_{out} = \{P_{out,pjob}\}$
RUN S=Open $Q = ()$ $J_{next} = 0$
Message $(P_{in,activation}, M_{in})$
S=Open
Message $(P_{in,job}, M_{in})$
$Q_{J_{next}}=M_{in}$
$J_{next}=J_{next}+1$
0
S=Closed
0
S=Open
DROP JOB
$Q=(Q_k)_{k=0}^{J_{next}-2}$
$J_{next}=J_{next}-1$
SEND JOB
Message $(P_{out,job}, Q_0)$
$Q=(Q_k)_1^{J_{next}-1}$
$J_{next}=J_{next}-1$
Message $(P_{in,deactivation}, M_{in})$
S=Closed

Fig. 15. Event relationship graph for the gate model



$P_{in} = \{P_{in,activation}, P_{in,deactivation}\}$
$P_{out} = \{P_{out,job}\}$
$RND_i \sim Exp(\lambda)$
RUN
S=Active
J=0
0
S=Active
BEGIN GENERATION
t_i=RND_i
S=Active
t_i
t_i
SEND JOB
J=J+1
Message ($P_{out,job}$,J)
S=Passive
0
Message ($P_{in,activation}$,M_{in})
0
0
Message ($P_{in,deactivation}$,M_{in})
S=Passive

Fig. 16. Event relationship graph for the generator model



$P_{in} = \{P_{in,job}\}$
$P_{out} = \{P_{out,k}\}_{k=0}^{n}$, $n \in N^+$
RUN
$P_{next} = 0$
Q = ( )
$J_{next} = 0$
Message ($P_{in,job}$,M_{in})
0
SEND JOB
$Q_{J_{next},P}=P_{out,P_{next}}$
$Q_{J_{next},M}=M_{in}$
$P_{next}=(P_{next}+1)$ % (n+1)
$J_{next}=J_{next}+1$
Message ($Q_{0,P}$,$Q_{0,M}$)
$Q=(Q_k)_{k=1}^{J_{next}-1}$
$J_{next}=J_{next}-1$

Fig. 17. Event relationship graph for the load balancer model

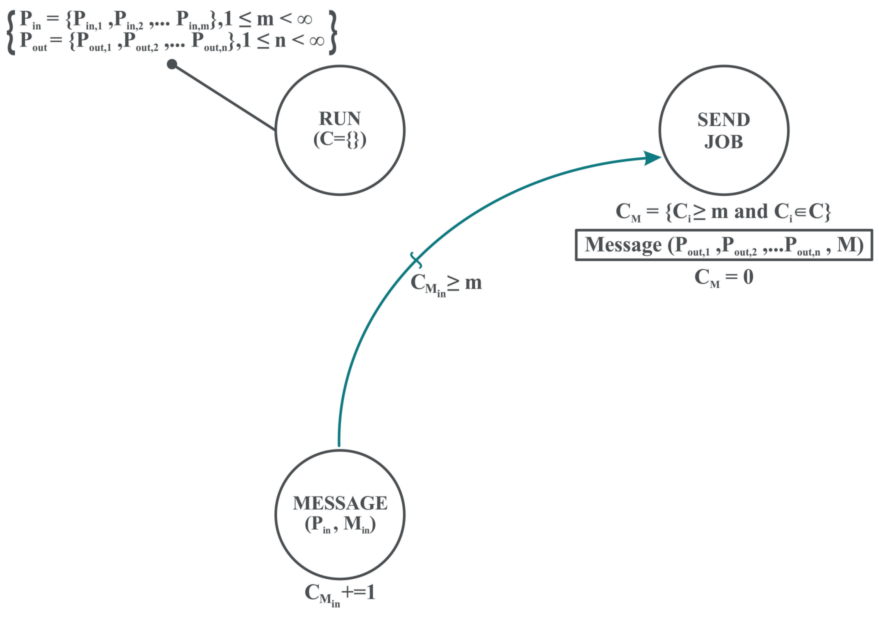

$\left\{ \begin{array}{l} P_{in} = \{P_{in,1}, P_{in,2}, \ldots P_{in,m}\}, 1 \le m < \infty \\ P_{out} = \{P_{out,1}, P_{out,2}, \ldots P_{out,n}\}, 1 \le n < \infty \end{array} \right\}$
RUN (C={})
SEND JOB
$C_M = \{C_i \ge m \text{ and } C_i \in C\}$
Message ($P_{out,1}, P_{out,2}, \ldots P_{out,n}$, M)
$C_M = 0$
$C_{M_{in}} \ge m$
MESSAGE (P_{in}, M_{in})
$C_{M_{in}}$+=1

Fig. 18. Event relationship graph for the parallel gateway model

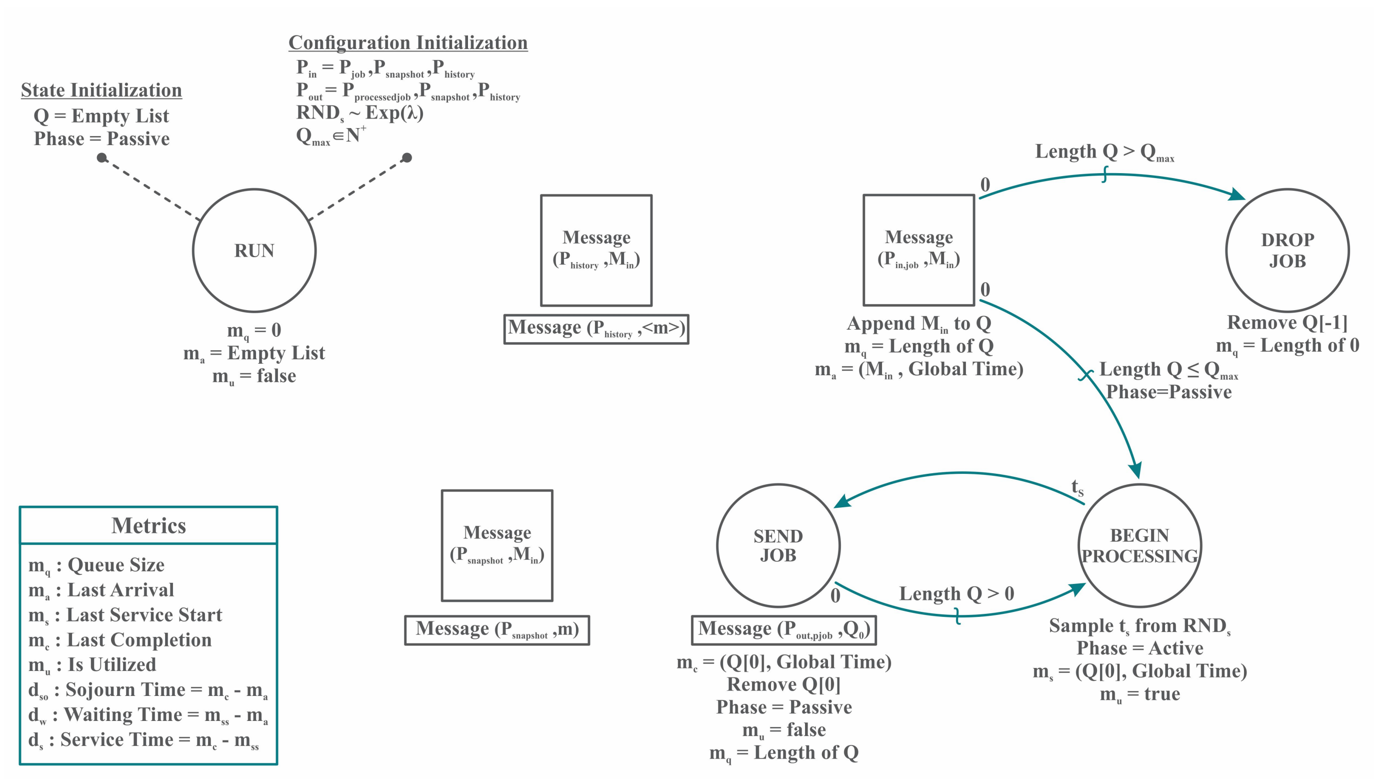

State Initialization
Q = Empty List
Phase = Passive
Configuration Initialization
$P_{in} = P_{job}, P_{snapshot}, P_{history}$
$P_{out} = P_{processedjob}, P_{snapshot}, P_{history}$
$RND_s \sim Exp(\lambda)$
$Q_{max} \in N^+$
RUN
$m_q = 0$
m_a = Empty List
m_u = false
Message ($P_{history}$, M_{in})
Message ($P_{history}$, <m>)
Message ($P_{in,job}$, M_{in})
Append M_{in} to Q
m_q = Length of Q
m_a = (M_{in}, Global Time)
Length Q > Q_{max}
0
0
DROP JOB
Remove Q[-1]
m_q = Length of 0
Length Q ≤ Q_{max}
Phase=Passive
Metrics
m_q : Queue Size
m_a : Last Arrival
m_s : Last Service Start
m_c : Last Completion
m_u : Is Utilized
d_{so} : Sojourn Time = m_c - m_a
d_w : Waiting Time = m_{ss} - m_a
d_s : Service Time = m_c - m_{ss}
Message ($P_{snapshot}$, M_{in})
Message ($P_{snapshot}$, m)
SEND JOB
t_s
BEGIN PROCESSING
Length Q > 0
0
Message ($P_{out,pjob}$, Q_0)
m_c = (Q[0], Global Time)
Remove Q[0]
Phase = Passive
m_u = false
m_q = Length of Q
Sample t_s from RND_s
Phase = Active
m_s = (Q[0], Global Time)
m_u = true

Fig. 19. Event relationship graph for the processor model

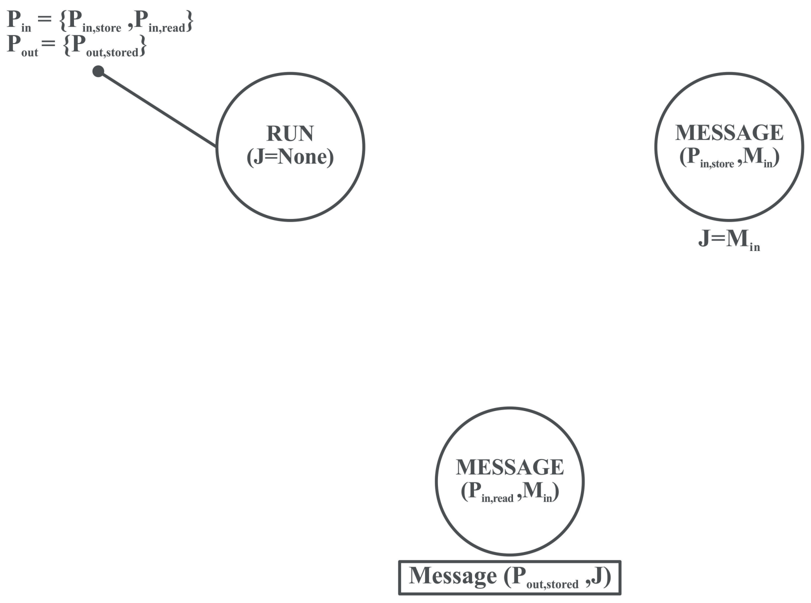
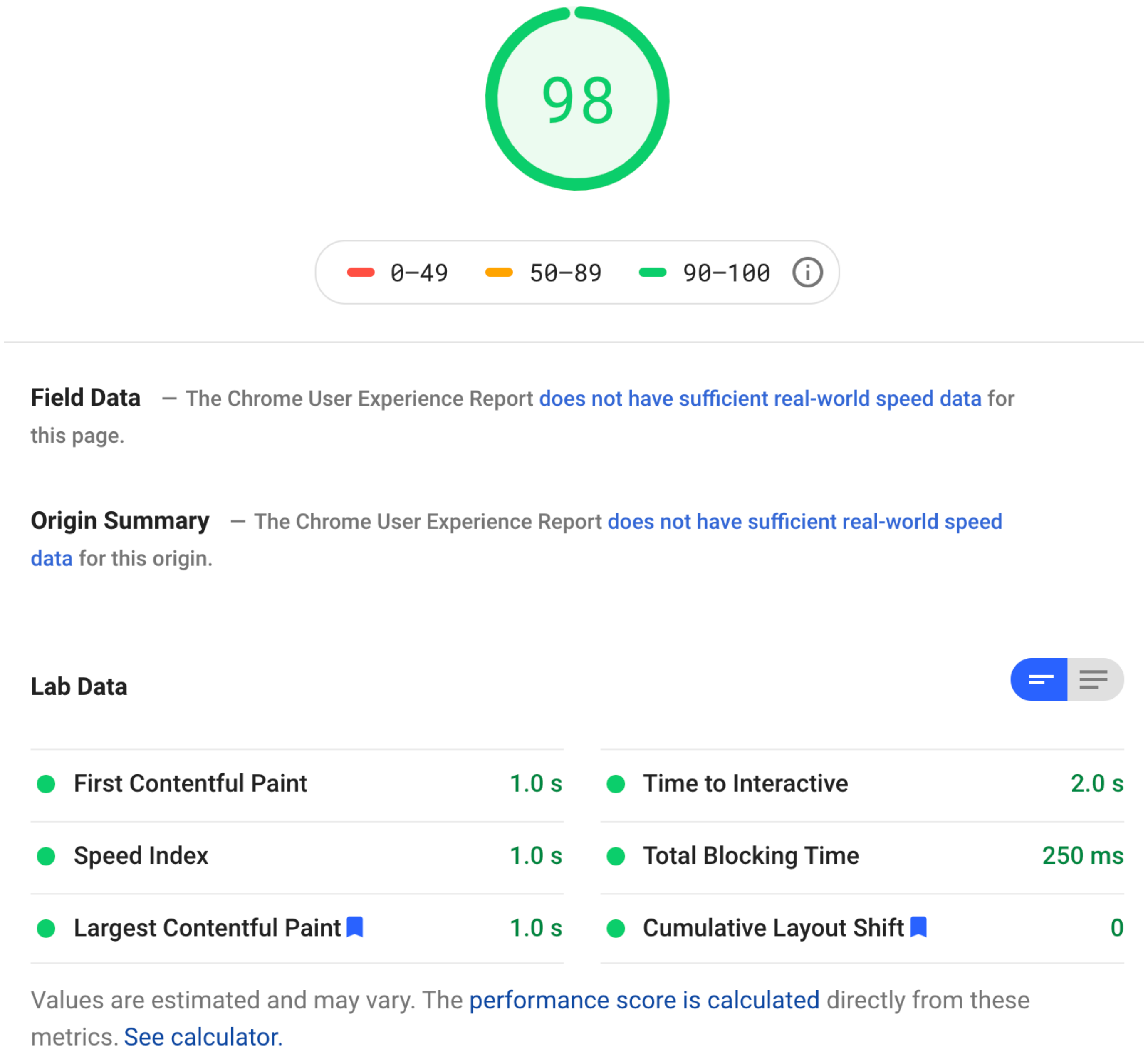

$P_{in} = \{P_{in,store}, P_{in,read}\}$
$P_{out} = \{P_{out,stored}\}$
RUN (J=None)
MESSAGE ($P_{in,store}$, M_{in})
$J=M_{in}$
MESSAGE ($P_{in,read}$, M_{in})
Message ($P_{out,stored}$, J)

Fig. 20. Event relationship graph for the storage model

Appendix B
Google Page Speed Results



98

🟥 0–49 🟧 50–89 🟩 90–100

Field Data — The Chrome User Experience Report does not have sufficient real-world speed data for this page.

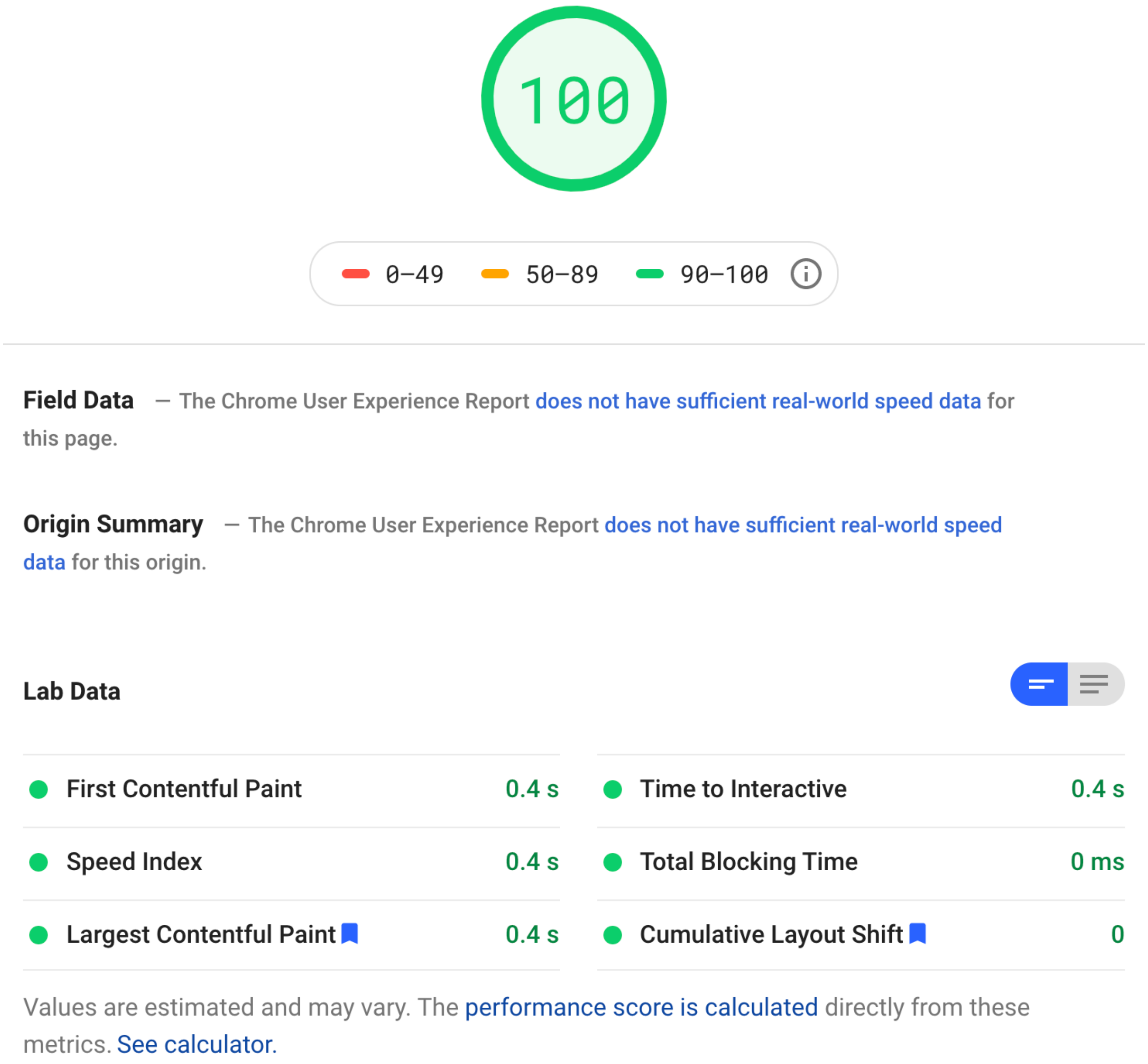
Origin Summary — The Chrome User Experience Report does not have sufficient real-world speed data for this origin.

Lab Data

●	First Contentful Paint	1.0 s
●	Speed Index	1.0 s
●	Largest Contentful Paint	1.0 s
●	Time to Interactive	2.0 s
●	Total Blocking Time	250 ms
●	Cumulative Layout Shift	0

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

Fig. 21. Google PageSpeed Insights mobile performance testing data



100

0–49 50–89 90–100

Field Data — The Chrome User Experience Report does not have sufficient real-world speed data for this page.

Origin Summary — The Chrome User Experience Report does not have sufficient real-world speed data for this origin.

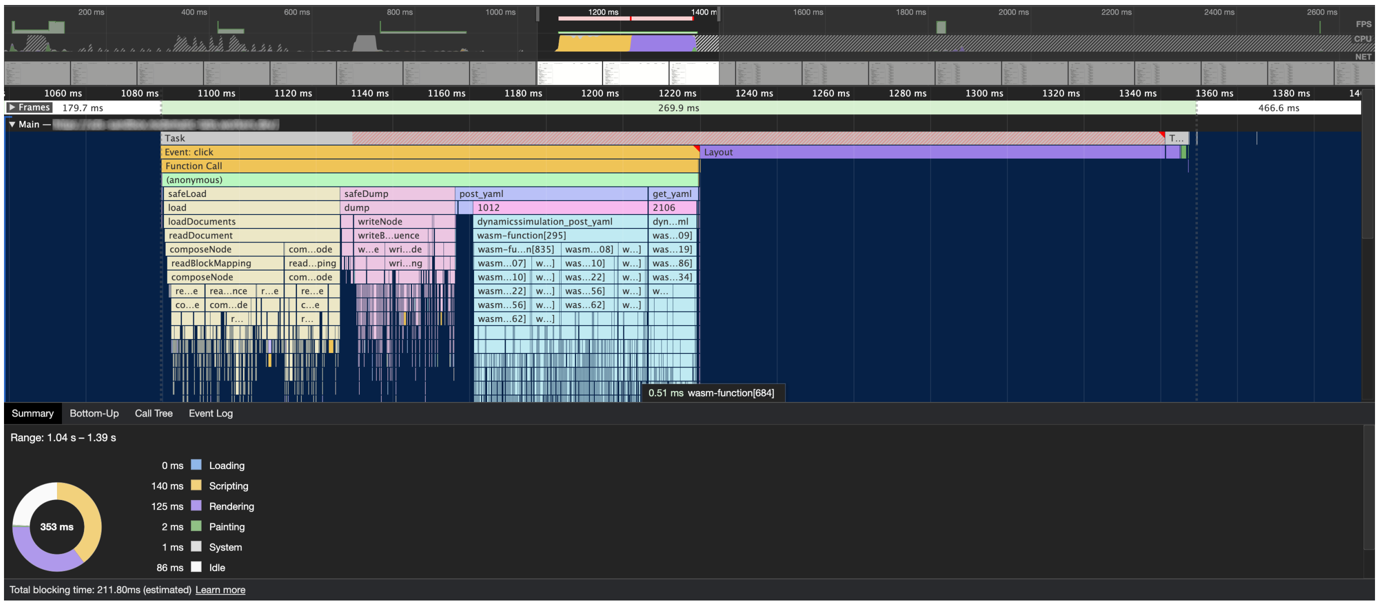
Lab Data

●	First Contentful Paint	0.4 s	●	Time to Interactive	0.4 s
●	Speed Index	0.4 s	●	Total Blocking Time	0 ms
●	Largest Contentful Paint	0.4 s	●	Cumulative Layout Shift	0

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

Fig. 22. Google PageSpeed Insights desktop performance testing data

Appendix C
Development Tools Screenshots



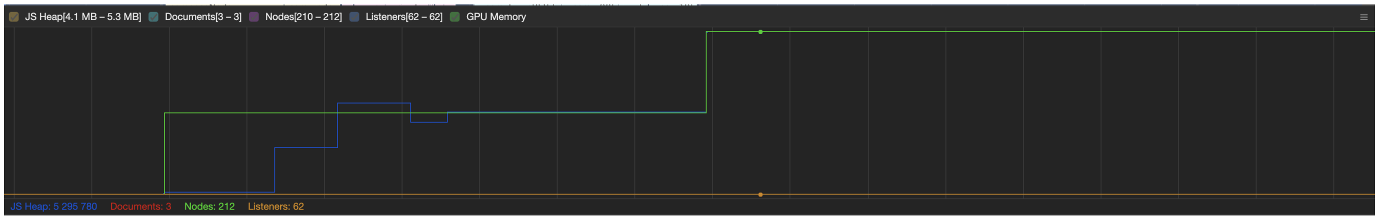
Range: 1.04 s – 1.39 s

0 ms	🟦	Loading
140 ms	🟨	Scripting
125 ms	🟪	Rendering
2 ms	🟩	Painting
1 ms	⬜	System
86 ms	⬜	Idle

353 ms

Total blocking time: 211.80ms (estimated) Learn more

Fig. 23. Initialization timing



JS Heap: 5 295 780 Documents: 3 Nodes: 212 Listeners: 62

Fig. 24. Initialization heap memory usage

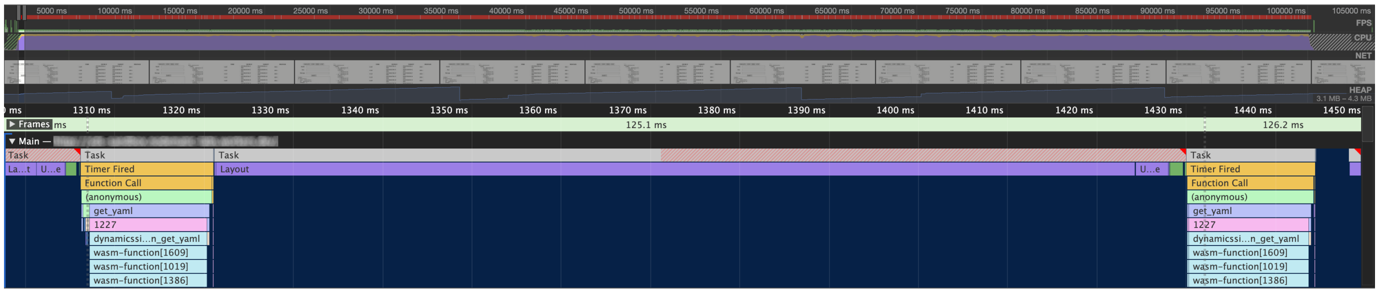


Fig. 25. Simulation step near the beginning of the simulation run

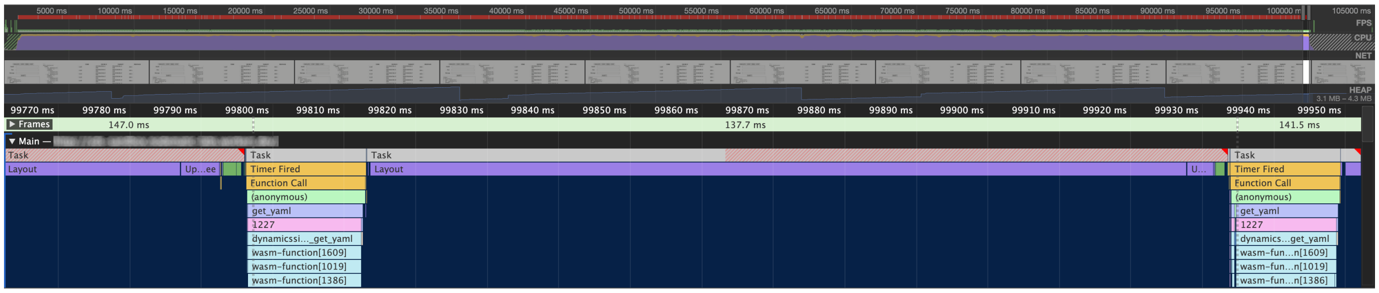


Fig. 26. Simulation step near the end of the simulation run

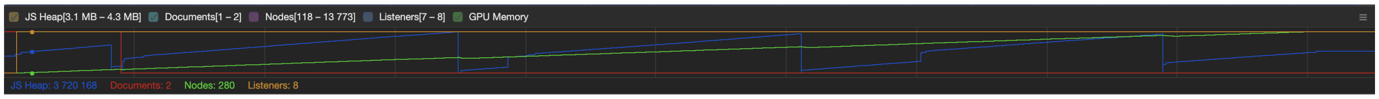


Fig. 27. Memory use during simulation execution