# Optimal resource allocation for tasks in Continuous Integration Systems

Authors:

- Stanislaw Swierc (stansw)
- Mateusz Machalica (stupaq)
- Lauri Myllari (lmyllari)
- Kevin McCormick (mckevin)

* @meta.com

https://github.com/StanislawSwierc/CCIW2023-8780

# Plan

1. Motivation
2. System Overview
3. Previous Solutions
4. Simulation Model
5. Maximize Success Rate
6. Minimize Total Computational Cost
7. Practical Considerations
8. Summary

# Motivation

- Meta operates Large Scale Continuous Integration system.
- Make the most efficient use of our compute capacity.
- Make it straightforward for developers to configure CI.

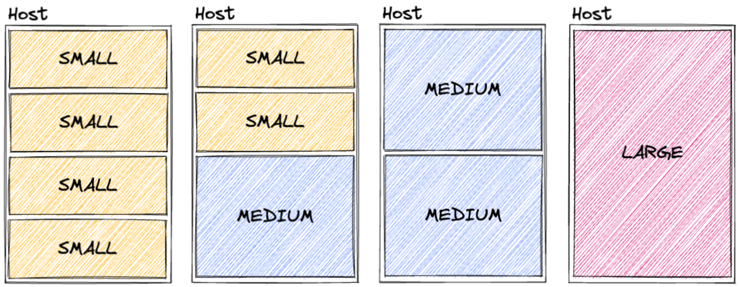
> *The goal of the Resource Allocation Module is to dispatch incoming tasks to worker queues to **minimize computational cost** while making sure all the tasks have enough resources to **succeed** in a in **timely manner**.*

# System Overview: Hosts and workers

A comprehensive description of the Resource Management System used at Meta Platforms Inc. is beyond the scope of this presentation. We will focus on a simplified version instead.

The simplified CI system consists of a homogeneous fleet of hosts, which are split into workers of three sizes:

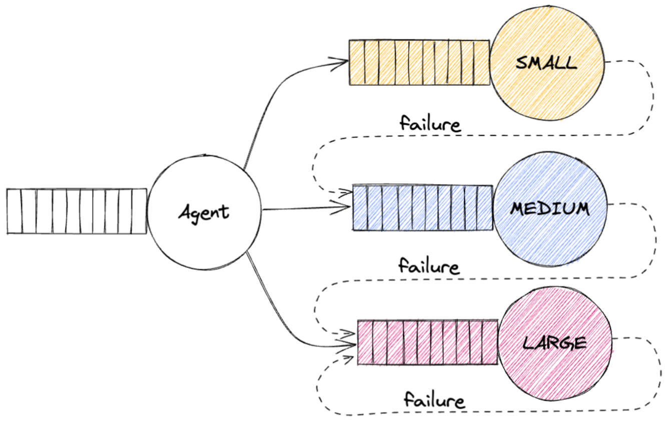
- Small (S) - 25% resources of a host
- Medium (M) - 50% resources of a host
- Large (L): 100% resources of a host



# System Overview: Queues and retry rules

When task arrives in the system it is initially processed by an agent which dispatches it to the appropriate queue.

If a task fails it gets retried on a larger worker until it reaches the largest worker.



# Previous Solutions

- **Empirical percentile**
    - Estimate percentile from historical data.
    - Good because computation is online.
    - Bad because selection of context can be hard.
- **Multivariate regression**
    - Train regression model on historical data.
    - Good because prediction is based on multiple inputs.
    - Bad because does not account for uncertainty.
- **Multivariate regression with uncertainty**
    - Train regression model with uncertainty on historical data (e.g. Monte Carlo Dropout).
    - Good because prediction is based on multiple inputs and accounts for uncertainty.
    - **Bad because it vulnerable to tasks which consume all the allocated resources**.

> *Tasks that consume all available memory will experience unbounded growth in their memory reservation, even if they could successfully run on smaller workers.*

# Previous Solutions: Greedy algorithms

> *Greedy algorithms serve as perhaps the simplest and most common approach to* ***online decision problems****. The following two steps are taken to generate each action:* ***(1) estimate a model from historical data*** *and* ***(2) select the action that is optimal for the estimated model****, breaking ties in an arbitrary manner. (source)*

# Simulation Model

This presentation is a Jupyter Notebook with a complete simulation code!

In [119]:

```
class Size(str, Enum):
    S = "SMALL"
    M = "MEDIUM"
    L = "LARGE"

@dataclass
class WorkerConfig:
    size: Size
    pred: Optional[str]
    succ: Optional[str]
    cost: float

class Processor:
    def run(self, size: Size) -> Tuple[float, bool]:
        return (1.0, True)

class Agent:
    def choose(self) -> Size:
        return Size.S

    def observe(self, size: Size, duration: float, success: bool):
        pass

    def report(self) -> Dict[str, Any]:
        return dict()
```

In [120]:

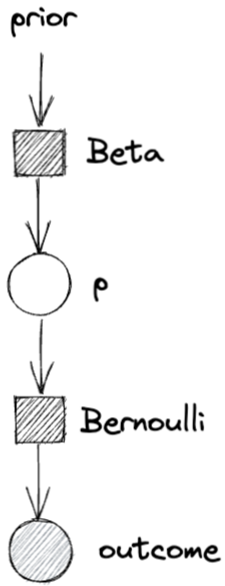
```
def simulate_minimalistic(workers, agent, processor, steps):
    for i in range(steps):
        attempt, retries = 1, 10

        size = agent.choose()

        while retries > 0:
            duration, success = processor.run(size)

            agent.observe(size, duration, success)

            if success:
                retries = 0
            elif workers[size].succ is None:
                retries = min(retries - 1, 1)
            else:
                size = workers[size].succ
                retries -= 1

            attempt += 1
```

# Maximize Success Rate: Statistical Model

We can model uncertainty with a small, hierarchical, generative model.

In [123]:

```
from scipy.stats import beta, bernoulli

@dataclass
class BetaParams:
    a: float
    b: float

@dataclass
class BernoulliParams:
    p: float

# https://en.wikipedia.org/wiki/Conjugate_prior
class BernoulliWithBetaPrior:
    def __init__(self, prior: BetaParams):
        self.hparams = prior

    def update(self, x: np.ndarray) -> None:
        a, b = astuple(self.hparams)
        n = np.size(x)
        s = np.sum(x.astype(bool))

        new_a = a + s
        new_b = b + n - s

        self.hparams = BetaParams(new_a, new_b)

    def sample_params(self) -> BernoulliParams:
        a, b = astuple(self.hparams)

        p = beta.rvs(a, b)

        return BernoulliParams(p)

    def estimate_params(self) -> BernoulliParams:
        a, b = astuple(self.hparams)

        p = beta.mean(a, b)

        return BernoulliParams(p)

    def sample_values(self) -> bool:
        params = self.sample_params()
        return bool(bernoulli.rvs(astuple(params)))
```



prior
Beta
p
Bernoulli
outcome

# Maximize Success Rate: Greedy

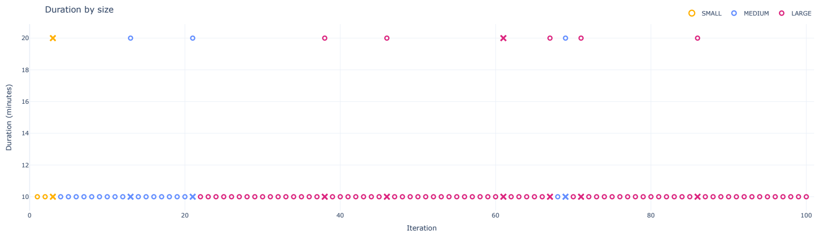
In [124]:

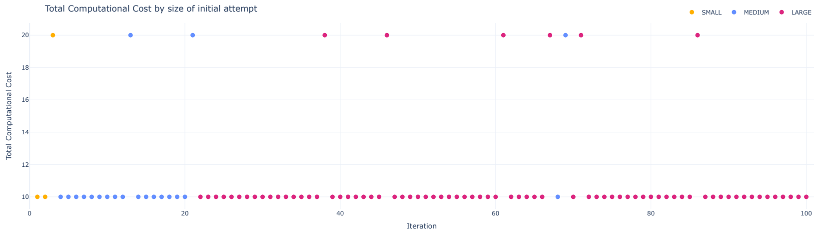
```
class GreedyAgent(Agent):
    def __init__(self, workers_config):
        self.success_models = {
            s: BernoulliWithBetaPrior(BetaParams(10, 1))
            for s in workers_config
        }

    def choose(self) -> Size:
        params = self.estimate_params()  # ⬅ Important!
        scores = self.score_params(params)
        return max(scores, key=scores.get)

    def estimate_params(self):
        return {
            s: m.estimate_params()
            for s, m in self.success_models.items()
        }

    def score_params(self, params):
        return {s: param.p for s, param in params.items()}

    def observe(self, size, duration, success):
        self.success_models[size].update(np.array([success]))
```

# Maximize Success Rate: Greedy

In [127]:

```
agent = GreedyAgent(INDEPENDENT_WORKERS_CONFIG)
processor = LognormBernoulliProcessor(SUCCESS_RATE_PROCESSOR_CONFIG)
executions = simulate(INDEPENDENT_WORKERS_CONFIG, agent, processor, 100
plot_duration_by_size(executions).show()
plot_total_cost_by_initial_attempt(executions).show()
```



# Maximize Success Rate: Epsilon-Greedy

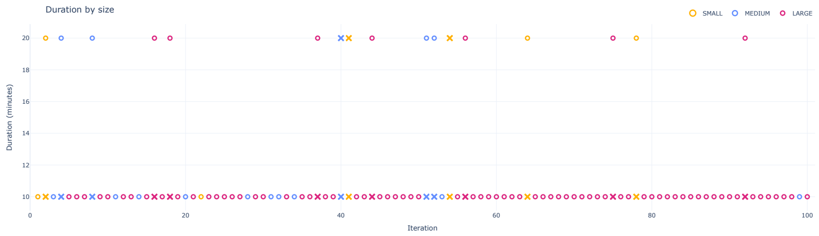
In [128]:

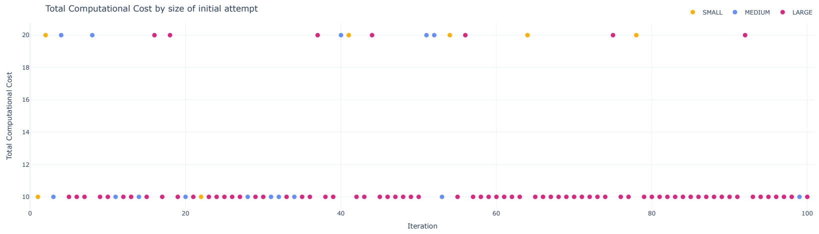
```
class EpsilonGreedyAgent(Agent):
    def __init__(self, workers_config, epsilon):
        self.epsilon = epsilon
        self.success_models = {
            s: BernoulliWithBetaPrior(BetaParams(10, 1))
            for s in workers_config
        }

    def choose(self) -> Size:
        if bernoulli.rvs(self.epsilon):  # ⬅ Important!
            return Size[np.random.choice(SIZE_NAMES)]
        else:
            params = self.estimate_params()
            scores = self.score_params(params)
            return max(scores, key=scores.get)

    def estimate_params(self):
        return {s: m.estimate_params() for s, m in self.success_models.

    def score_params(self, params):
        return {s: param.p for s, param in params.items()}

    def observe(self, size: Size, duration: float, success: bool) -> No
        self.success_models[size].update(np.array([success]))
```

# Maximize Success Rate: Epsilon-Greedy

In [129]:

```
agent = EpsilonGreedyAgent(INDEPENDENT_WORKERS_CONFIG, epsilon=0.3)
processor = LognormBernoulliProcessor(SUCCESS_RATE_PROCESSOR_CONFIG)
executions = simulate(INDEPENDENT_WORKERS_CONFIG, agent, processor, 100
plot_duration_by_size(executions).show()
plot_total_cost_by_initial_attempt(executions).show()
```



# Maximize Success Rate: Thompson Sampling

> *Thompson sampling is an algorithm for online decision problems where **actions are taken sequentially** in a manner that must balance between **exploiting what is known** to maximize immediate performance and **investing to accumulate new information** that may improve future performance.*

> *Thompson sampling offers flexible, elegant and efficient approach to exploration in a wide range of **structured decision problems**.*

**Algorithm 3** Greedy$(\mathcal{X}, p, q, r)$

**for** $t = 1, 2, \dots$ **do**
  #estimate model:
  $\hat{\theta} \leftarrow \mathbb{E}_p[\theta]$

  #select and apply action:
  $x_t \leftarrow \mathrm{argmax}_{x \in \mathcal{X}} \mathbb{E}_{q_{\hat{\theta}}}[r(y_t) | x_t = x]$
  Apply $x_t$ and observe $y_t$

  #update distribution:
  $p \leftarrow \mathbb{P}_{p,q}(\theta \in \cdot | x_t, y_t)$
**end for**

**Algorithm 4** Thompson$(\mathcal{X}, p, q, r)$

**for** $t = 1, 2, \dots$ **do**
  #sample model:
  Sample $\hat{\theta} \sim p$

  #select and apply action:
  $x_t \leftarrow \mathrm{argmax}_{x \in \mathcal{X}} \mathbb{E}_{q_{\hat{\theta}}}[r(y_t) | x_t = x]$
  Apply $x_t$ and observe $y_t$

  #update distribution:
  $p \leftarrow \mathbb{P}_{p,q}(\theta \in \cdot | x_t, y_t)$
**end for**

Russo, Daniel J., Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. "A tutorial on thompson sampling." Foundations and Trends® in Machine Learning 11, no. 1 (2018): 1-96.

# Maximize Success Rate: Thompson Sampling

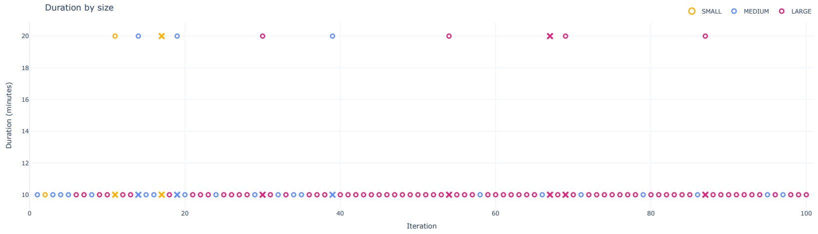
In [130]:
```
class GreedyAgent(Agent):
    def __init__(self, workers_config):
        self.success_models = {
            s: BernoulliWithBetaPrior(BetaParams(2, 2))
            for s in workers_config
        }

    def choose(self) -> Size:
        params = self.estimate_params()  # ⬅ Important!
        scores = self.score_params(params)
        return max(scores, key=scores.get)

    def estimate_params(self):
        return {
            s: m.estimate_params()
            for s, m in self.success_models.items()
        }

    def score_params(self, params):
        return {s: param.p for s, param in params.items()}

    def observe(self, size, duration, success):
        self.success_models[size].update(np.array([success]))
```

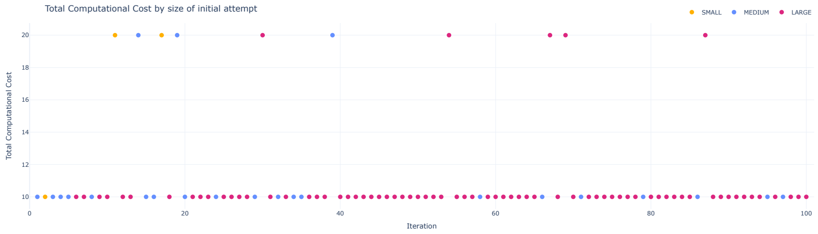
In [131]:
```
class SuccessRateTSAgent(Agent):
    def __init__(self, workers_config):
        self.success_models = {
            s: BernoulliWithBetaPrior(BetaParams(2, 2))
            for s in workers_config
        }

    def choose(self) -> Size:
        params = self.sample_params()  # ⬅ Important!
        scores = self.score_params(params)
        return max(scores, key=scores.get)

    def sample_params(self):
        return {
            s: m.sample_params()
            for s, m in self.success_models.items()
        }

    def score_params(self, params):
        return {s: param.p for s, param in params.items()}

    def observe(self, size, duration, success):
        self.success_models[size].update(np.array([success]))
```

# Maximize Success Rate: Thompson Sampling

In [133]:

```
agent = SuccessRateTSAgentWithReporting(INDEPENDENT_WORKERS_CONFIG)
processor = LognormBernoulliProcessor(SUCCESS_RATE_PROCESSOR_CONFIG)
executions = simulate(INDEPENDENT_WORKERS_CONFIG, agent, processor, 100
plot_duration_by_size(executions).show()
plot_total_cost_by_initial_attempt(executions).show()
```

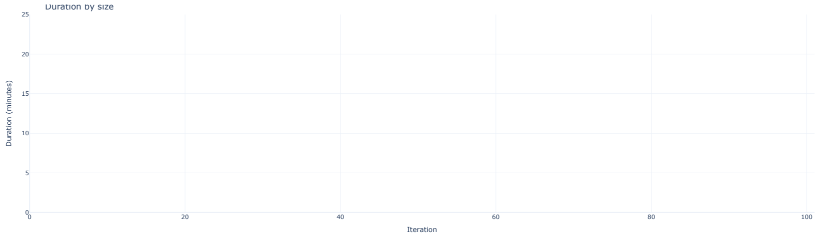


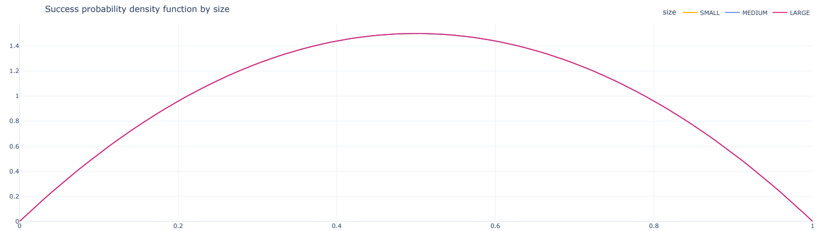



# Maximize Success Rate: Thompson Sampling

In [135]:

```
frame = 1
plot_duration_by_size(executions[:frame]).update_layout(yaxis_range=[0,
plot_success_dist(executions[:frame]).show()
```

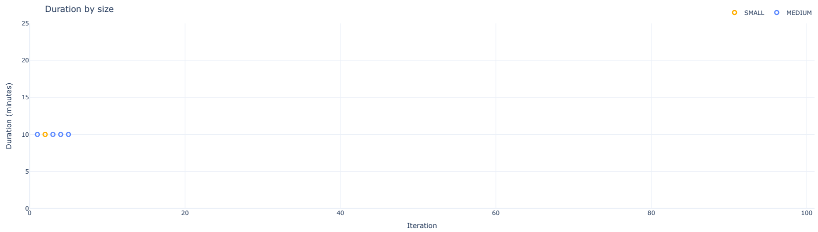


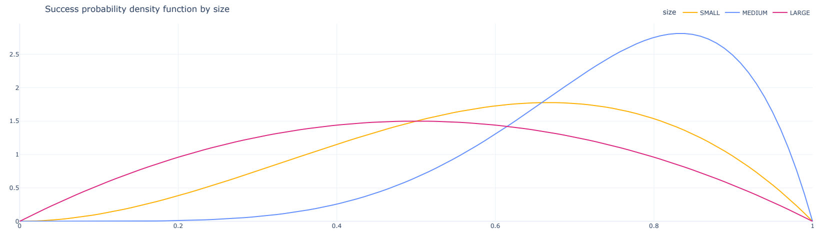



# Maximize Success Rate: Thompson Sampling

In [136]:

```
frame = 6
plot_duration_by_size(executions[:frame]).update_layout(yaxis_range=[0,
plot_success_dist(executions[:frame]).show()
```

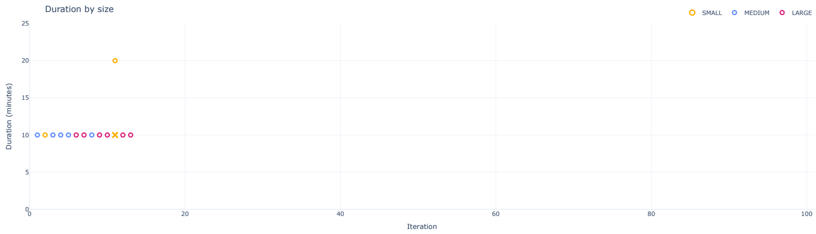


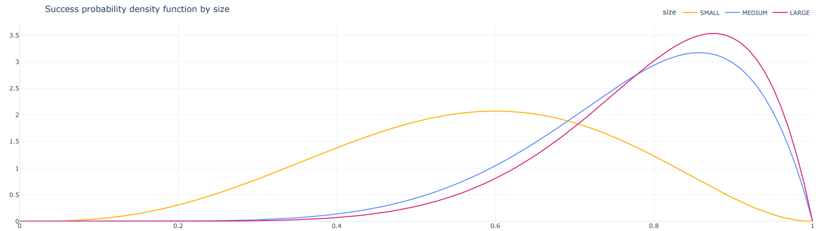



# Maximize Success Rate: Thompson Sampling

In [137]:

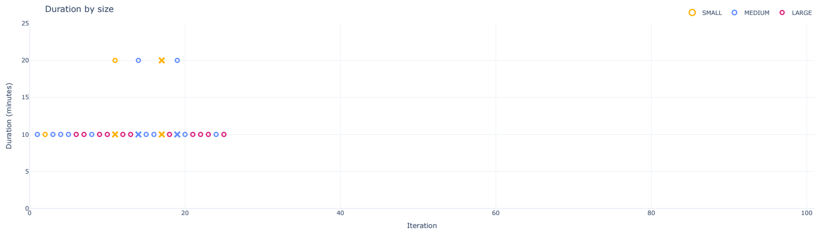
```
frame = 15
plot_duration_by_size(executions[:frame]).update_layout(yaxis_range=[0,
plot_success_dist(executions[:frame]).show()
```

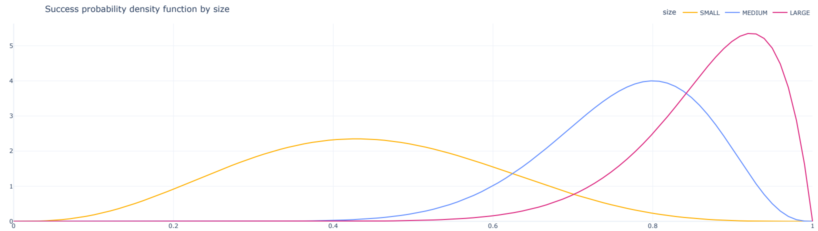


# Maximize Success Rate: Thompson Sampling

In [138]:
```
frame = 30
plot_duration_by_size(executions[:frame]).update_layout(yaxis_range=[0,
plot_success_dist(executions[:frame]).show()
```

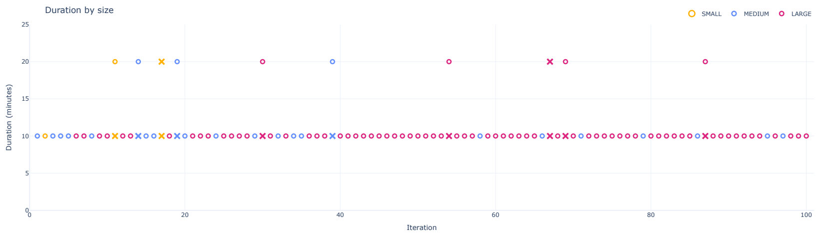


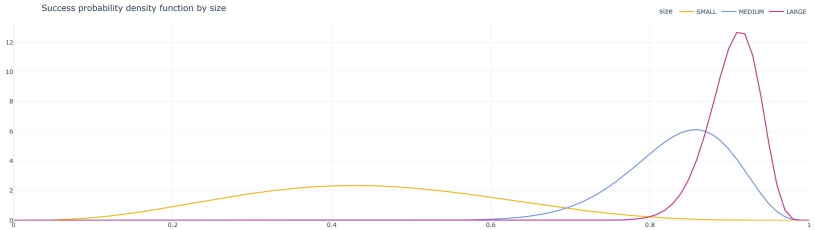



# Maximize Success Rate: Thompson Sampling

In [139]:

```
plot_duration_by_size(executions).update_layout(yaxis_range=[0, 25]).sh
plot_success_dist(executions).show()
```

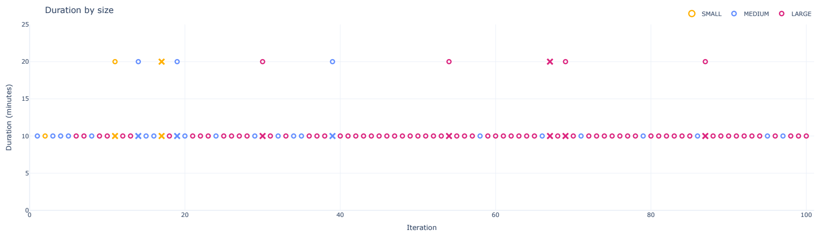


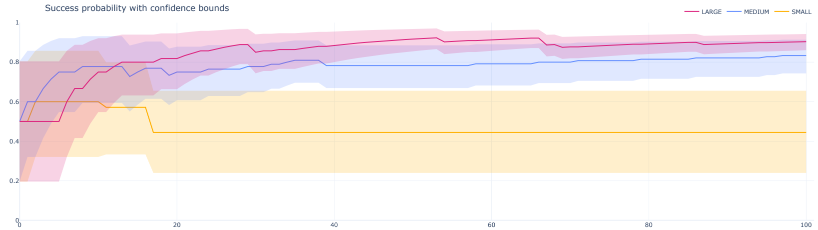



# Maximize Success Rate: Thompson Sampling

In [140]:

```
plot_duration_by_size(executions).update_layout(yaxis_range=[0, 25]).sh
plot_success_confidence(executions).show()
```



# Minimize Total Computational Cost: Objective

> *Computational Cost of the first attempt and all the subsequent retries.*

$$
\begin{gathered}
\mathbb{E}[total\_computational\_cost(t, w)] = \\
\mathbb{E}[duration(t, w)] \cdot cost(w) \\
+ (1 - \mathbb{E}[success(t, w)]) \\
\cdot \mathbb{E}[total\_computational\_cost(t, next(w))]
\end{gathered}
$$

Statistical model:

$$ln(duration) \sim \mathcal{N}(\mu, \sigma^2)$$

$$success \sim Bernoulli(p)$$

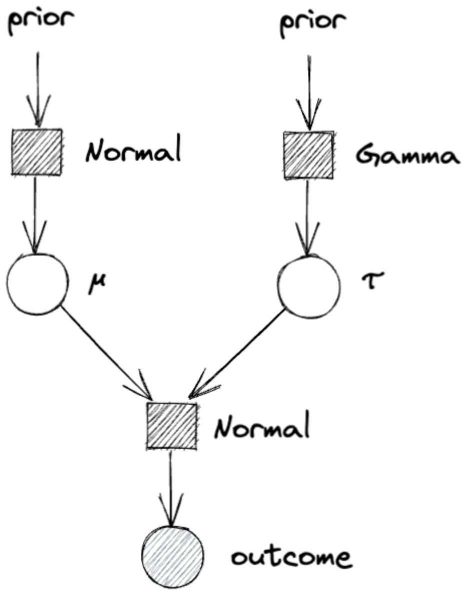
Considers:

- Duration
- Success rate
- Cost of workers
- Retry rules

# Minimize Total Computational Cost: Model

In [141]:

```
from scipy.stats import norm, gamma

@dataclass
class NormParams:
    mu: float
    tau: float

@dataclass
class NormalGammaParams:
    alpha: float
    beta: float
    mu0: float
    n0: float

# https://en.wikipedia.org/wiki/Conjugate_prior
class NormWithNormalGammaPrior:
    def __init__(self, prior: NormalGammaParams):
        self.hparams = prior

    def update(self, x: np.ndarray) -> None:
        alpha, beta, mu0, n0 = astuple(self.hparams)
        n = np.size(x)
        x_avg = np.mean(x)
        x_var = np.sum(np.square(x - x_avg)) / n

        new_alpha = alpha + n / 2
        new_beta = beta + n * x_var / 2 + (n * n0) * np.square(x_avg -
        new_mu0 = (n * x_avg + n0 * mu0)/ (n + n0)
        new_n0 = n0 + n

        self.hparams = NormalGammaParams(new_alpha, new_beta, new_mu0,

    def sample_params(self) -> BernoulliParams:
        alpha, beta, mu0, n0 = astuple(self.hparams)

        tau = gamma.rvs(alpha, scale=1 / beta)
        mu = norm.rvs(loc=mu0, scale=np.sqrt(1 / (n0 * tau)))

        return NormParams(mu, tau)

    def estimate_params(self) -> BernoulliParams:
        alpha, beta, mu0, n0 = astuple(self.hparams)

        tau = gamma.mean(alpha, scale=1 / beta)
        mu = norm.mean(loc=mu0, scale=np.sqrt(1 / (n0 * tau)))

        return NormParams(mu, tau)

    def sample_values(self) -> bool:
        params = self.sample_params()
        return norm.rvs(loc=params.mu, scale=np.sqrt(1 / params.tau))
```


prior
prior
Normal
Gamma
μ
τ
Normal
outcome

# Minimize Total Computational Cost: Expected Value

In [142]:

```python
def calculate_total_computational_cost(
    workers_config: List[WorkerConfig],  # Sorted topologically!
    success_params: BetaParams,
    duration_params: NormParams,
):
    total_cost = {}
    for worker in reversed(workers_config):
        p_success = success_params[worker.size].p
        duration_mean = np.exp(
            duration_params[worker.size].mu +
            1 / (2 * duration_params[worker.size].tau)
        )

        # Limit exploration of workers with low success rates.
        if p_success < 0.8:
            p_success = 0.1

        # Cost of the first attempt.
        cost_initial_attempt = worker.cost * duration_mean

        # Cost of retrying failed task possibly on a larger worker.
        total_cost_retry = (
            total_cost[worker.succ]
            if worker.succ is not None
            else cost_initial_attempt
        )

        total_cost[worker.size] = cost_initial_attempt + (1 - p_success

    return total_cost
```

# Minimize Total Computational Cost: Thompson Sampling

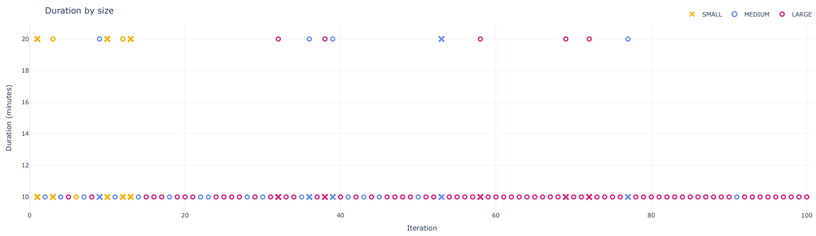
In [143]:

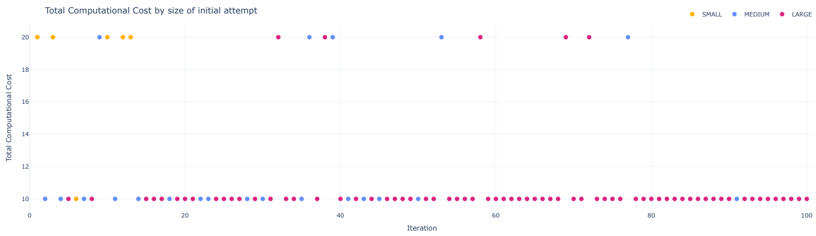
```
class TotalcalComputationalCostTSAgent(Agent):
    def __init__(self, workers_config):
        self.workers_config = workers_config
        self.success_models = {
            s: BernoulliWithBetaPrior(BetaParams(1, 1))
            for s in workers_config
        }
        self.duration_models = {
            s: NormWithNormalGammaPrior(NormalGammaParams(2.0, 1.0, np.
            for s in workers_config
        }

    def choose(self):
        params = self.sample_params()
        scores = self.score_params(params)
        return min(scores, key=scores.get)

    def sample_params(self):
        return (
            {s: model.sample_params() for s, model in self.success_mode
            {s: model.sample_params() for s, model in self.duration_mod
        )

    def observe(self, size, duration, success):
        self.success_models[size].update(np.array([success]))
        self.duration_models[size].update(np.array([np.log(duration)]))

    def score_params(self, params):
        success_params, duration_params = params
        return calculate_total_computational_cost(
            self.workers_config.values(),
            success_params,
            duration_params,
        )
```

# Minimize Total Computational Cost: Thompson Sampling

In [149]:

```
agent = TotalcalComputationalCostTSAgent(INDEPENDENT_WORKERS_CONFIG)
processor = LognormBernoulliProcessor(SUCCESS_RATE_PROCESSOR_CONFIG)
executions = simulate(INDEPENDENT_WORKERS_CONFIG, agent, processor, 100
plot_duration_by_size(executions).show()
plot_total_cost_by_initial_attempt(executions).show()
```

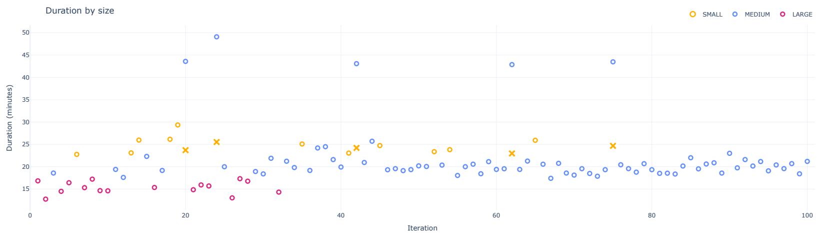


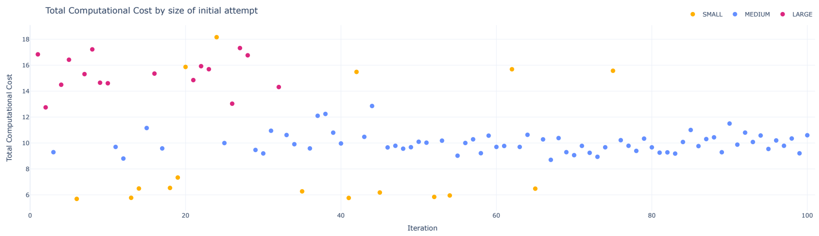



# Minimize Total Computational Cost: Thompson Sampling

In [146]:

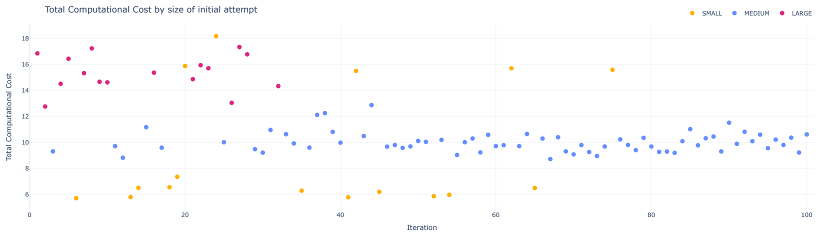
```
agent = TotalComputationalCostTSAgentWithReporting(WORKERS_CONFIG)
processor = LognormBernoulliProcessor(PROCESSOR_CONFIG)
executions = simulate(WORKERS_CONFIG, agent, processor, 100, seed=10)
plot_duration_by_size(executions).show()
plot_total_cost_by_initial_attempt(executions).show()
```

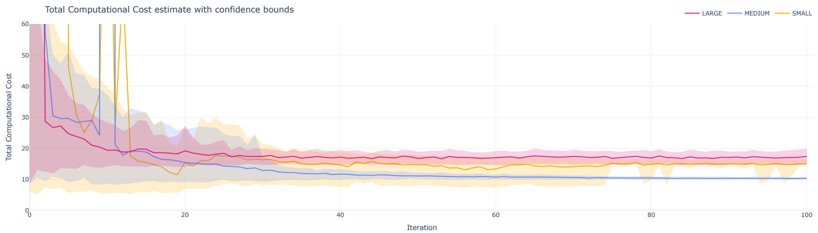


# Minimize Total Computational Cost: Thompson Sampling

In [148]:
```
plot_total_cost_by_initial_attempt(executions).show()
plot_total_cost_confidence(executions).update_layout( yaxis_range=[0, 6
```



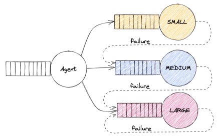



# Practical Considerations

- **Souce of entropy** - stochastic algorithms require good sources of entropy.
  - Use good soruce of entropy to generate seed and save it in task definition.
- **Nonstationary systems** - success rate can change over time.
  - Ignore historical observations made beyond N time periods in the past.
- **Conditional dependency** - duration depends on the outcome.
  - Use different models for success and failure outcomes.
- **Unnecessary exploration** - if task keeps failing on M there is no point in exploring S.
  - Allow the model to only explore sizes neighboring to the current best size.
- **Multidimensional problem space** - workers differ on more than one dimension.
  - Switch from a list to a graph of options using successor relationship.
- **Shared state** - results of initial attempt can be partially cached.
  - Allow agent to observe results only from the initial attempt.
- **Dynamic contexts** - task identifier changes over time.
  - Use a hierarchy of tasks and fallback if the number of observations is low
- **Conflicting objectives** - some tasks should optimize for speed.
  - Create a way to override allocation decisions.

# Summary

- Greedy algorithms are vulnerable to tasks which consume all the allocated resources.
- Resource allocation module should optimize for the Total Computational Cost defined as the cost of the first attempt and all the subsequent retries.
- Thompson Sampling algorithm offer asymptotically optimal solutions to the problem of minimizing Total Computational Cost. Its efficient approach to exploration in structured decision problems makes it a great fit for complex retry rules.



# Resouces


1. Russo, Daniel J., Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. "A tutorial on thompson sampling." Foundations and Trends® in Machine Learning 11, no. 1 (2018): 1-96.
2. Marmerola G.D., "Introduction to Thompson Sampling: the Bernoulli bandit" (2017)
3. Jordan, Michael I. "The conjugate prior for the normal distribution." Lecture notes on Stat260: Bayesian Modeling and Inference (2010).