

A Cost-Conscious eCommerce Architecture for Small Business

How a Composable Commerce Platform Can Reduce Operating Overhead Without Sacrificing Functionality, Security, or a Path to Scale



A Technical and Business Whitepaper

Prepared for: Business Owners, eCommerce Store Owners, and Small-Business Operators

Architecture examined: Warm Presents eCommerce Platform

Research date: September 2026

Executive Summary

For a small business, an eCommerce website is not simply a website.

It is a sales channel, product catalog, inventory system, payment system, shipping desk, customer communication system, order-management system, and increasingly, an automation platform.

Yet many small businesses are asked to accept a recurring technology cost simply to keep those capabilities available.

The business challenge behind the Warm Presents build was straightforward: the owners did not want to spend \$39 or more every month simply to maintain their website, while also absorbing the separate costs associated with payment processing, shipping, and other commerce services.

The objective therefore was **not to build the cheapest website possible**.

It was to determine whether a modern eCommerce business could have the functionality expected from a much more expensive hosted commerce platform while keeping recurring infrastructure costs controlled, avoiding unnecessary subscriptions, maintaining security, and preserving a credible path to future growth.

The resulting architecture takes a **composable commerce** approach.

The customer-facing storefront is separated from the commerce engine. Medusa v2 manages products, inventory, carts, orders, customers, authentication, and commerce operations. PostgreSQL provides durable transactional storage. Redis provides shared infrastructure for caching, event handling, workflow execution, rate limiting, and other state that should not live only in application memory. Railway hosts the backend infrastructure, while Netlify hosts the storefront. Supabase Storage provides object/file storage. Stripe handles payment processing. Shippo provides shipping rates, labels, and tracking. Resend provides transactional email.

This separation is important from a business perspective.

The business is not required to purchase one giant platform containing every possible feature whether it needs those features or not. Instead, individual capabilities can be provided by specialized services and connected through APIs.

That creates a different economic model:

Pay for the infrastructure and services the business actually uses, rather than paying a large fixed platform fee simply for access to a bundle of capabilities.

The architecture also deliberately automates work that would otherwise consume the owner's time.

When an order is placed, fulfillment can occur automatically. Shipping rates can be calculated through Shippo. A shipping label can be purchased automatically. Customers can receive transactional emails. Tracking events can update the order lifecycle. Inventory changes can trigger storefront cache invalidation. Payment failures can generate notifications. Customers can cancel eligible orders themselves.

These are not merely technical conveniences.

For a small business, **automation is a labor-cost reduction strategy.**

The architecture therefore attempts to optimize five business variables simultaneously:

1. **Recurring operating cost**
2. **Owner and staff time**
3. **Customer experience**
4. **Security and reliability**
5. **Future scalability**

The result is not a “cheap website.” It is a **purpose-built commerce platform designed around the economics of a small business.**

In production, the platform’s hosting infrastructure currently costs **about \$9 per month** — Railway’s Hobby plan hosts the backend, PostgreSQL, and Redis, while the storefront runs on Netlify’s free plan — compared with the \$39 monthly platform fee that prompted the project.

Table of Contents

How a Composable Commerce Platform Can Reduce Operating Overhead Without Sacrificing Functionality, Security, or a Path to Scale	1
.....	1
Executive Summary	2
Table of Contents.....	3
1. The Business Problem	5
The Real Cost of Running an Online Store	5
2. The Economic Principle: Fixed Cost vs. Usage Cost	6
.....	8
3. The Architecture in Business Terms	9

4. The Storefront: Selling Without Making Every Page a Database Transaction	10
5. The Commerce Engine: The Store Is More Than a Website	11
6. PostgreSQL: The Business Record	11
7. Redis: More Than a Cache	12
8. Shipping: Turning a Manual Task Into an Automated Process	13
9. The Safety Net: Manual Shipping Remains Available.....	14
10. Payments: Outsource the High-Risk Financial Infrastructure	14
11. Customer Communication: Email as an Automated Business Function.....	15
12. Reducing Owner Work Through Automation.....	16
13. Security Designed Around the Business	17
14. Resilience: What Happens When Something Breaks?	18
Provider integrations are conditional	18
Shipping has a fallback.....	18
Redis provides shared infrastructure	18
Storefront changes are event-driven	18
15. Storefront Updates Without Rebuilding Everything Manually	19
16. Why This Can Be Less Expensive Without Being “Cheap”	19
17. The Current Cost Philosophy.....	20
18. Why Not Simply Use an All-in-One Commerce Platform?	21
19. The Hidden Advantage: Avoiding Vendor Lock-In.....	23
20. Scalability: Built for Growth, Not for Imaginary Scale	24
21. Deliberate Simplicity	25
22. What the Business Owner Actually Gets	25
Sell products	26
Manage inventory.....	26
Accept payments	26
Calculate shipping	26
Purchase shipping labels.....	26
Track shipments.....	26
Communicate with customers	26
Protect customer accounts.....	26
Protect order information.....	26
Recover from provider problems.....	26

Update the storefront automatically	26
Grow later	27
23. The Owner's Time Is a Business Asset	27
24. Risks and Trade-Offs	28
Engineering complexity	28
Operational responsibility	28
Multiple providers	28
Cost can increase with success	28
Current deployment limitations	28
25. The Development Process	29
26. Testing and Quality	30
27. The Path Forward	30
Near term	31
Medium term	31
Growth stage	31
28. What This Build Demonstrates	31
A small business does not necessarily need a small-business-grade technology platform.	32
29. The Larger Business Lesson	32
Conclusion	33
Appendix A — Current Architecture at a Glance	35
Appendix B — Current Known Limitations	35
Completed / No Longer Open	35
Remaining / Deliberately Deferred	36
Appendix C — Sources and Research Notes	36

1. The Business Problem

The Real Cost of Running an Online Store

A business owner generally thinks about an online store differently from an engineer.

The owner asks:

- What does it cost me every month?
- How much does it cost when I make a sale?
- How much work do I have to do?
- Will customers be able to check out?
- Will they receive their orders?
- Will the site be available when people want to buy?
- Can I add products?
- Can I change prices?
- Can I eventually grow without rebuilding everything?
- What happens if one service goes down?
- Am I locked into one company?

These questions are more important to the owner than the specific technology underneath the site.

The traditional answer has frequently been an all-in-one hosted commerce platform.

That approach has significant advantages. The platform provider manages much of the infrastructure and gives the business a unified administrative experience.

But the convenience comes with a trade-off: the business pays for the platform continuously, regardless of whether the business is using every component included in that platform.

For a small or early-stage merchant, that fixed expense can be meaningful.

The Warm Presents project began from a different premise:

Can modern cloud infrastructure and open-source commerce software be assembled so that the business only pays recurring infrastructure costs where those costs create actual business value?

That question drove many of the architectural decisions.

2. The Economic Principle: Fixed Cost vs. Usage Cost

One of the most important ideas behind the architecture is separating **platform cost** from **business activity cost**.

Some costs should exist regardless of whether the store receives an order.

Examples include:

- Domain registration
- Core application infrastructure
- Database infrastructure
- Basic hosting

Other costs naturally increase when the business becomes more successful:

- Payment processing
- Shipping labels
- Shipping-rate requests
- Email volume
- Storage consumption
- Bandwidth
- Compute consumption

That distinction is useful.

If a store sells nothing during a slow month, it should not necessarily incur the same variable commerce costs as a store processing hundreds of orders.

Stripe's current standard U.S. pricing, for example, is transaction-based rather than requiring a monthly payment-platform subscription: its published standard rate is 2.9% + \$0.30 per successful domestic card transaction.

Shippo similarly provides an API model in which its Starter tier currently includes the first 30 labels per month and then charges per label, with additional fees for services such as rating, tracking, delivery estimates, and address validation.

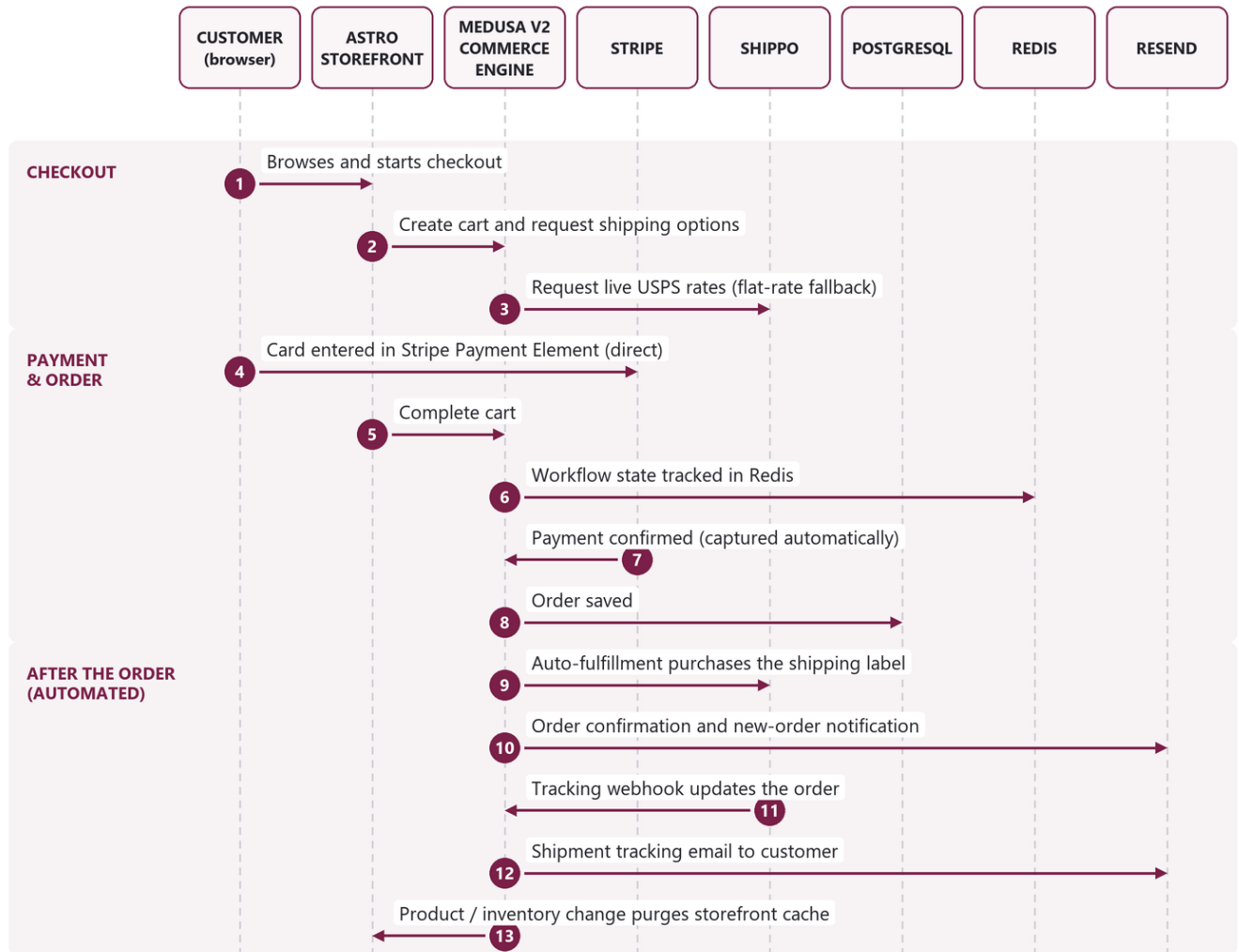
Resend currently provides a free tier for transactional email and paid plans as email volume grows.

The significance is not that these particular services are always the cheapest option.

The significance is the **economic pattern**:

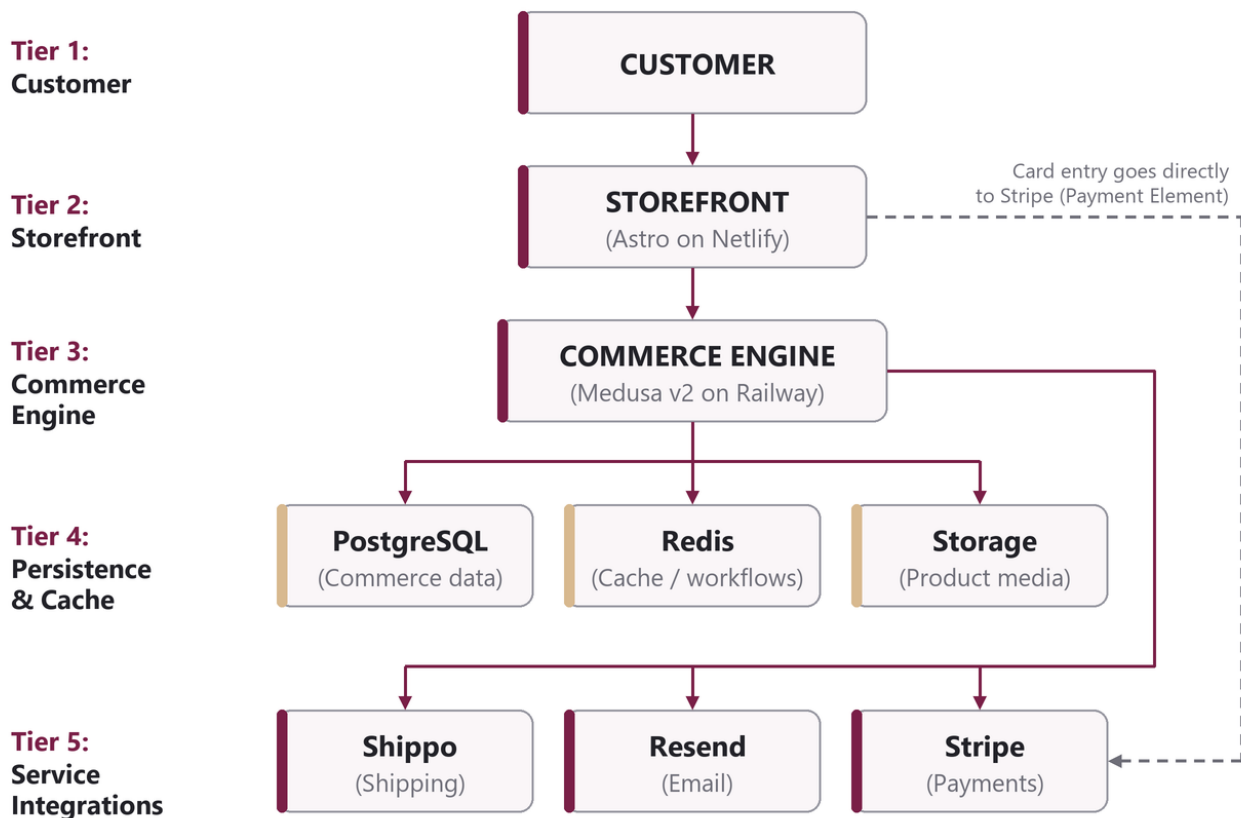
The business can connect specialized services whose costs generally grow alongside business activity.

That is fundamentally different from paying a large fixed platform fee simply to have access to the underlying capability.



3. The Architecture in Business Terms

At a high level, the platform can be understood as five business layers.



The commerce engine calls each service; card details never pass through it.

The important point for a business owner is that **each component has a job**.

The storefront exists to sell.

Medusa exists to operate the store.

PostgreSQL exists to preserve business data.

Redis exists to make the application faster and coordinate work.

Stripe exists to process payments.

Shippo exists to handle shipping infrastructure.

Resend exists to communicate with customers.

This separation allows each part to evolve independently.

4. The Storefront: Selling Without Making Every Page a Database Transaction

The customer-facing portion of the system uses Astro.

Astro's architecture emphasizes rendering the majority of a page as lightweight HTML while loading JavaScript only where interaction requires it. Astro describes this as its “Islands Architecture.”

For an eCommerce business, that matters because most of the customer's first experience with a store is not inherently transactional.

A product page needs to:

- Display the product
- Show images
- Explain the product
- Communicate price
- Present purchasing options
- Provide search-engine-friendly content
- Load quickly

Only portions of that experience need constant interactivity.

The shopping cart, customer account functions, checkout interactions, and other dynamic features can communicate with the commerce backend.

This creates an important business advantage:

The marketing and merchandising experience does not need to carry the entire weight of the commerce backend.

That separation also makes it possible to optimize the storefront for speed, SEO, content presentation, and conversion without forcing those concerns into the same application responsible for inventory and financial transactions.

5. The Commerce Engine: The Store Is More Than a Website

The backend uses Medusa v2.

According to Medusa's architecture documentation, a Medusa application separates HTTP/API routes, workflows, modules, and the underlying PostgreSQL data layer. Infrastructure modules can provide caching, events, files, notifications, and workflow execution.

For the business owner, this means the backend becomes the operational system behind the storefront.

The project documentation identifies Medusa as the system responsible for:

- Products
- Inventory
- Carts
- Orders
- Customers
- Authentication
- Store API
- Admin API

That distinction is crucial.

The storefront is what customers see.

Medusa is what the business operates.

This means that changing the visual storefront does not require replacing the underlying commerce system.

Likewise, changing a shipping provider does not require rebuilding the storefront.

That is one of the primary advantages of the architecture.

6. PostgreSQL: The Business Record

PostgreSQL serves as the durable database layer.

This is where the system needs dependable persistence for commerce information.

For a business, the database is ultimately holding information such as:

- Products
- Customers
- Orders
- Inventory
- Commerce relationships
- Transaction-related state

Medusa's current architecture documentation identifies PostgreSQL as its standard underlying datastore.

The business value is straightforward:

The store's core business data is not trapped inside the storefront.

That provides a foundation for future applications as well.

The same commerce backend could eventually support:

- A mobile application
- A wholesale portal
- Additional storefronts
- Marketplace integrations
- Custom reporting
- Business automation
- Additional sales channels

The storefront is therefore only one customer of the commerce platform.

7. Redis: More Than a Cache

Redis is often described as a caching technology.

In this build, its role is considerably broader.

The project uses Redis for:

- Event bus infrastructure
- Workflow engine
- Caching
- Login protection
- Rate limiting

Medusa specifically recommends the Redis workflow engine for production use, because workflows track transactions and their steps and can support reliable rollback behavior.

This has a direct operational benefit.

Suppose a customer places an order.

That order can trigger multiple activities:

1. Payment processing
2. Order creation
3. Fulfillment
4. Shipping-label purchase
5. Email notification
6. Inventory updates
7. Tracking
8. Storefront cache updates

Those activities should not necessarily be treated as one giant request.

Separating the work allows the system to become more resilient and easier to scale.

For the owner, the result is simple:

The system can perform more work automatically without requiring the owner to perform each step manually.

8. Shipping: Turning a Manual Task Into an Automated Process

Shipping is one of the places where eCommerce platforms can create substantial operational overhead.

The Warm Presents architecture integrates Shippo as a custom Medusa fulfillment provider.

The implementation supports:

- Live USPS rate shopping
- Address validation
- Shipping estimates
- Label purchase
- Tracking

- Tracking webhooks
- Automatic fulfillment

Most importantly, fulfillment can occur automatically after an order is placed.

The system can purchase the shipping label without requiring the owner to open the administration panel and manually initiate fulfillment.

That changes the economics of the store.

The owner is no longer paying for software simply to give themselves a sophisticated interface for manually repeating tasks that software can perform automatically.

The software becomes the employee performing the repetitive portion of the workflow.

9. The Safety Net: Manual Shipping Remains Available

Automation does not mean removing every manual option.

The architecture deliberately keeps a standard flat-rate shipping option.

The project documentation explains that this fallback is intentionally maintained so a Shippo outage does not completely prevent checkout.

This is an important architectural principle:

Automate the normal path while preserving a viable recovery path.

A small business does not necessarily need an infinitely sophisticated system.

It needs a system that continues to function when one component has a problem.

10. Payments: Outsource the High-Risk Financial Infrastructure

Stripe handles payment processing.

This is another deliberate division of responsibility.

The store does not need to become a payment processor.

Stripe provides the payment infrastructure while the commerce backend manages the business relationship between the cart, order, customer, and payment.

The project also implements:

- Automatic payment capture
- Failed-payment handling
- Stripe webhook verification
- Saved payment methods
- A custom save-card-at-checkout solution

Stripe's current standard pricing is transaction-based, with no monthly fee under its standard pricing model.

Again, the important business concept is not the exact percentage.

It is that **payment infrastructure scales with sales activity**.

When the business sells more, it pays more because it is processing more transactions.

That is a much more natural cost relationship than paying a large fixed fee simply because a payment system exists.

11. Customer Communication: Email as an Automated Business Function

Transactional email is handled through Resend.

The system sends messages for events including:

- Order confirmation
- Internal new-order notifications
- Payment failures
- Order cancellations
- Shipment tracking
- Label purchases

- Password resets

This is more important than it may initially appear.

A customer does not experience “Resend.”

They experience:

“I placed an order and immediately received confirmation.”

“My order shipped and I received tracking information.”

“My payment failed and I was informed.”

“I requested a password reset and received the email.”

The architecture therefore converts technical events into customer-service events.

Resend currently offers a free transactional-email tier and paid tiers as volume increases.

That again supports the usage-oriented economic model.

12. Reducing Owner Work Through Automation

The biggest economic advantage of the architecture may not actually be hosting cost.

It may be **time**.

Consider what happens when an order is placed.

A conventional small-business workflow might require the owner or employee to:

1. Notice the order
2. Verify payment
3. Prepare fulfillment
4. Calculate shipping
5. Purchase a label
6. Send confirmation
7. Send tracking
8. Update order status

9. Manage inventory
10. Handle exceptions

The Warm Presents implementation automates substantial portions of that workflow.

The project includes automatic fulfillment, shipping-label creation, transactional notifications, tracking events, order cancellation, payment-failure handling, and storefront cache updates.

The business implication is significant:

The platform is not merely reducing software expense. It is reducing administrative labor.

For a small business, that may be considerably more valuable than saving a few dollars on hosting.

13. Security Designed Around the Business

A low-cost architecture is only valuable if it is safe.

The project therefore does not equate “low cost” with “minimal security.”

Security controls include:

- CORS restrictions
- JWT authentication
- Signed cookies
- Guest-order ownership checks
- Login brute-force protection
- reCAPTCHA escalation
- Redis-backed rate limiting
- Stripe webhook verification
- Protected order cancellation
- Security headers
- Separate storage credentials
- Secret-based webhook protection

Guest order lookup is particularly important.

A customer can access their order without necessarily having a full account, but the API requires an ownership check or matching order email and intentionally returns the same not-found behavior when authorization fails.

That is the type of detail that can easily be overlooked in a small-business build.

It is also the type of detail that matters tremendously once real customer information is involved.

14. Resilience: What Happens When Something Breaks?

No online business infrastructure is immune to failure.

The question is therefore not:

“Can this system ever fail?”

The better question is:

“What happens when one part fails?”

The architecture incorporates several resilience strategies.

Provider integrations are conditional

Stripe, Shippo, and Resend are conditionally registered based on their configuration rather than making the entire application fail simply because an optional provider is unavailable.

Shipping has a fallback

A manual shipping option remains available.

Redis provides shared infrastructure

This avoids relying exclusively on application-local memory for cache, workflow, event, and rate-limit state.

Storefront changes are event-driven

Product, inventory, and collection changes trigger the appropriate cache purge or rebuild behavior.

The business benefit is reduced operational fragility.

15. Storefront Updates Without Rebuilding Everything Manually

One of the more interesting aspects of the architecture is the interaction between the commerce backend and the storefront.

When inventory or product information changes, the backend can trigger cache invalidation.

When collections change, the backend can trigger a Netlify rebuild.

Netlify's build-hook system supports triggering builds programmatically and can optionally clear the build cache.

This creates a useful separation:

The commerce system remains authoritative about commerce data while the storefront remains optimized for customer delivery.

The business owner therefore does not need to think about manually synchronizing two systems.

The architecture handles the synchronization.

16. Why This Can Be Less Expensive Without Being “Cheap”

It is important to distinguish **cost-conscious engineering** from **under-engineering**.

Under-engineering means removing capabilities to save money.

Cost-conscious engineering means identifying which capabilities genuinely require recurring infrastructure and selecting an appropriate service for each.

This project does the latter.

For example:

- PostgreSQL is used because commerce data requires durable storage.
- Redis is used because shared cache/workflow/event infrastructure is useful.
- Stripe is used because payment processing should be delegated to a specialized provider.
- Shippo is used because shipping infrastructure is complicated and naturally usage-based.
- Resend is used because reliable transactional email is not worth maintaining internally.
- Supabase Storage is used for object storage rather than placing product media inside the application itself.
- Netlify handles the storefront delivery/build layer.
- Railway hosts the backend infrastructure.

Supabase's current storage model is also usage-oriented: storage is measured by GB-hours, with included quota depending on plan and overage charges based on usage.

The architecture therefore avoids reinventing infrastructure that is inexpensive to consume but expensive to build and maintain.

17. The Current Cost Philosophy

The monthly cost of this system depends on actual usage and provider plans, so it is best understood as a collection of cost categories.

Cost Category	Economic Behavior
Domain	Generally fixed annual cost
Storefront hosting	Fixed/usage-based depending on hosting plan
Backend hosting	Base subscription + resource usage
PostgreSQL	Infrastructure/resource cost
Redis	Infrastructure/resource cost
Object storage	Usage-based
Payment processing	Primarily transaction-based
Shipping API	Primarily usage-based

Cost Category	Economic Behavior
Shipping labels	Per shipment
Email	Volume-based
Development/maintenance	Human labor
Domain-specific features	Development cost

Railway's current pricing illustrates the model clearly: its plans combine a base subscription with resource usage, and the Hobby plan currently starts at \$5/month with \$5 of included resource usage.

Netlify currently offers a \$0 Free plan and paid plans beginning at \$9/month, with usage credits covering categories such as production deploys, compute, bandwidth, and web requests.

These figures are **current published provider pricing**; the actual bill for any store depends on the selected plans and actual consumption.

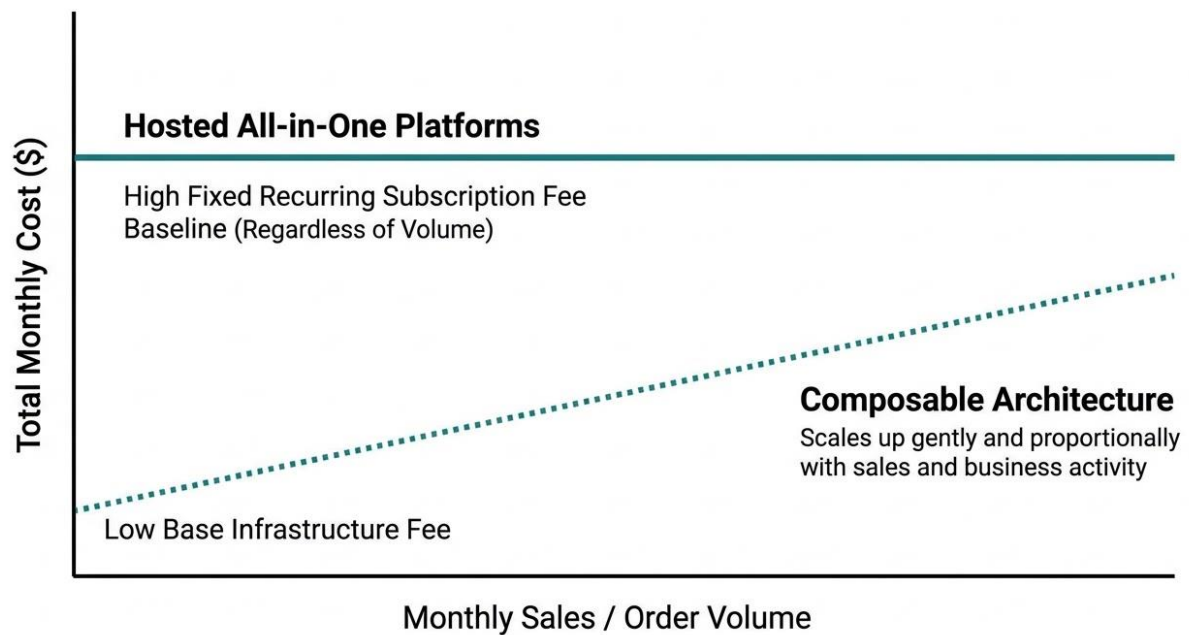
For reference, the Warm Presents production environment currently runs on **about \$9 per month** in hosting infrastructure: Railway's Hobby plan hosts the backend, PostgreSQL, and Redis, and the storefront runs on Netlify's free plan. That figure excludes transaction-based costs such as Stripe payment processing and Shippo shipping labels, and it will change as plans and usage change.

18. Why Not Simply Use an All-in-One Commerce Platform?

An all-in-one platform is not inherently wrong.

For many businesses, it may be the appropriate choice.

The trade-off is primarily one of **control, recurring cost, customization, and operational responsibility**.



Consideration	Hosted Commerce Platform	Composable Architecture
Initial setup	Usually easier	More engineering required
Recurring platform fee	Usually fixed	Distributed across services
Customization	Platform-dependent	Very high
Infrastructure control	Limited	Greater
Vendor dependence	Higher	Distributed
Operations	Mostly outsourced	More responsibility
Automation	Platform capabilities	Customizable
Data architecture	Vendor-defined	Business-defined
Scaling	Usually built in	Architected explicitly
Developer involvement	Lower	Higher
Small-business ownership	Simpler	Greater control

The key question is therefore not:

“Which architecture is better?”

It is:

“Which operating model makes sense for this business?”

For an owner who wants maximum simplicity and is comfortable paying a recurring platform fee, an all-in-one platform can make sense.

For an owner who wants control over the storefront, commerce logic, integrations, and long-term operating model—and is willing to invest in engineering—a composable architecture provides significantly more control.

The Warm Presents project deliberately selected the latter model.

19. The Hidden Advantage: Avoiding Vendor Lock-In

Another business advantage is optionality.

Medusa's architecture explicitly treats modules as replaceable components. Its documentation notes that infrastructure modules can be replaced with alternative implementations.

That principle appears directly in this build.

Shippo is a module.

Resend is a module.

Storage is a provider.

Redis is infrastructure.

Stripe is a payment provider.

The storefront is independent of the backend.

This means a future business decision could be:

“Shippo is no longer the best fit.”

The answer does not have to be:

“Rebuild the store.”

It can instead be:

“Replace the fulfillment integration.”

That is an important distinction for a growing business.

20. Scalability: Built for Growth, Not for Imaginary Scale

The system was not designed around the assumption that Warm Presents would immediately become a massive retailer.

That would itself be wasteful.

Instead, it provides architectural escape routes.

Today, the backend can run as a server and worker arrangement. The project configuration explicitly supports separating those responsibilities.

That matters because increasing commerce activity does not necessarily require rebuilding the application.

Future scaling could involve:

- More backend resources
- Additional application replicas
- Dedicated worker capacity
- More database resources
- Redis scaling
- More aggressive storefront caching
- Additional shipping carriers
- Additional payment capabilities
- Search infrastructure
- Read replicas
- Additional sales channels

Medusa's modular architecture is specifically intended to allow infrastructure and domain integrations to be replaced or extended as requirements change.

The objective is therefore not “infinite scale.”

It is **appropriate scale**.

21. Deliberate Simplicity

One of the strongest characteristics of the build is what it does **not** attempt to implement.

The current project deliberately defers:

- UPS/FedEx integration
- Return-label automation
- Full Content Security Policy implementation
- CI/CD automation

This is good engineering from a business perspective.

Every feature has a cost.

A feature costs money to develop, test, secure, maintain, document, and eventually support.

Building something simply because “an eCommerce platform should have it” can be wasteful.

The better question is:

Does this feature currently create enough business value to justify its lifecycle cost?

If not, defer it.

That keeps the platform aligned with the business rather than the other way around.

22. What the Business Owner Actually Gets

The technical architecture ultimately produces a set of business capabilities.

The owner gets a store that can:

Sell products

The storefront presents products and communicates with the commerce backend.

Manage inventory

Inventory lives in the commerce platform rather than being maintained manually in disconnected systems.

Accept payments

Stripe handles payment processing.

Calculate shipping

Shippo provides live shipping rates.

Purchase shipping labels

Fulfillment can automatically generate labels.

Track shipments

Shipping events feed back into the commerce system.

Communicate with customers

Transactional email handles order, payment, password, and shipment communications.

Protect customer accounts

Login brute-force protection with reCAPTCHA escalation, Redis-backed rate limiting, and JWT and signed-cookie authentication protect customer accounts.

Protect order information

Guest-order authorization prevents arbitrary order access.

Recover from provider problems

Manual shipping remains available.

Update the storefront automatically

Commerce events trigger cache invalidation and rebuilds.

Grow later

The commerce engine, database, infrastructure, and storefront are not locked into one monolithic application.

23. The Owner's Time Is a Business Asset

Small-business technology decisions often focus heavily on dollars.

They should also focus on **minutes**.

If an owner spends five minutes manually handling every order and processes 200 orders per month, that represents approximately 16.7 hours of work.

If automation removes most of that repetitive work, the economic value may be larger than the monthly hosting savings.

This is why the architecture should be evaluated in terms of **Total Cost of Ownership**, not merely hosting.

A more useful equation is:

Total Cost of Ownership = Infrastructure + SaaS + Transaction Fees + Maintenance + Human Time + Opportunity Cost

The architecture attacks several of those variables simultaneously.

It reduces unnecessary subscriptions.

It automates repetitive work.

It keeps transaction costs tied to actual commerce activity.

It separates infrastructure so individual components can be changed.

And it provides a path to scale without requiring an immediate enterprise architecture.

24. Risks and Trade-Offs

The architecture is not free of trade-offs.

Engineering complexity

A custom commerce system requires engineering expertise.

A hosted commerce platform transfers much of that responsibility to the vendor.

Operational responsibility

The business now owns more of its technology decisions.

Backups, deployment procedures, monitoring, credentials, dependency updates, and infrastructure configuration become important.

Multiple providers

Using multiple services reduces dependence on one vendor but increases the number of relationships that must be managed.

Cost can increase with success

Usage-based infrastructure is an advantage at modest scale, but high traffic and transaction volume can create substantial costs.

That is not necessarily a failure.

A business processing significantly more sales should naturally consume more infrastructure.

Current deployment limitations

The project currently has no repository-level CI pipeline, and deployment verification remains manual.

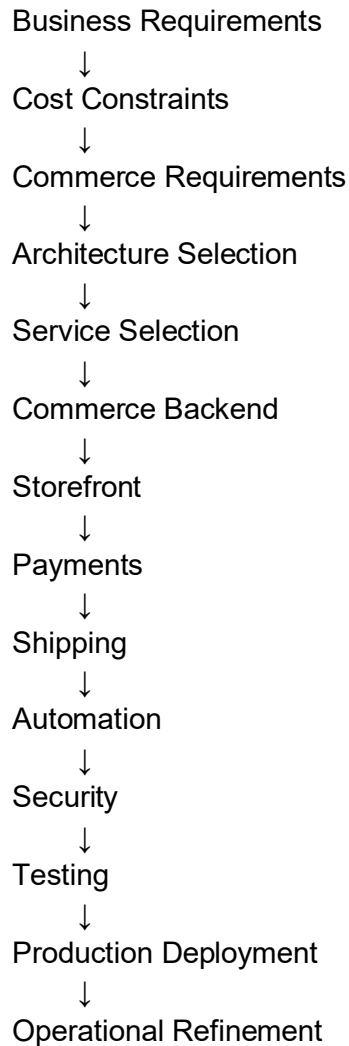
This should eventually be addressed as operational maturity increases.

25. The Development Process

The build process was not simply:

Design a website → write code → launch.

It was closer to:



The implementation also demonstrates iterative engineering.

For example, the project documentation records changes such as:

- Moving from SMTP-based email to Resend because of production environment constraints

- Correcting Stripe capture behavior
- Adding automatic fulfillment
- Adding custom shipping-estimate behavior
- Adding guest-order authorization
- Adding rate limiting
- Adding cache invalidation
- Removing an unconditional weekly rebuild once event-driven rebuilds were working

This illustrates an important principle:

The architecture was allowed to evolve based on real operational requirements rather than theoretical completeness.

26. Testing and Quality

The project is backed by 138 automated backend tests covering the order, payment, and fulfillment pipeline. HTTP integration tests also exist for storefront and order-lifecycle flows, although those require a reachable PostgreSQL environment to execute.

This matters because automation without testing can simply automate mistakes.

A business-critical eCommerce platform must be particularly careful around:

- Payments
- Orders
- Inventory
- Fulfillment
- Customer authorization
- Shipping
- Refunds

Those are financial and customer-service operations.

Testing therefore becomes part of the business protection strategy.

27. The Path Forward

The architecture already provides a foundation for future improvements.

Near term

Finish the remaining production-readiness items:

- Finish populating accurate weights and dimensions for every product
- Final payment-method verification
- Final webhook verification
- Security review

Medium term

Improve operational maturity:

- Automated CI
- Automated deployment verification
- Monitoring and alerting
- Better backup/recovery procedures
- Content Security Policy
- Additional shipping providers if business demand warrants them

Growth stage

Introduce capabilities only when business volume justifies them:

- Advanced search
- Additional fulfillment providers
- More sophisticated analytics
- Additional sales channels
- Wholesale functionality
- Customer segmentation
- Advanced marketing automation
- Additional application workers
- Database scaling
- Read replicas

The important point is that **growth should drive architectural expansion**.

The business should not pay the complexity cost of infrastructure it does not yet need.

28. What This Build Demonstrates

The Warm Presents project demonstrates a broader idea that is particularly relevant to small businesses:

A small business does not necessarily need a small-business-grade technology platform.

It needs a **right-sized technology platform**.

There is a meaningful difference.

A small business can use:

- A serious commerce engine
- A real relational database
- Production-grade caching
- Workflow infrastructure
- Professional payment processing
- Automated shipping
- Transactional email
- API-driven integrations
- Automated fulfillment
- Authentication controls
- Rate limiting
- Event-driven cache management

without necessarily accepting the economics of a large all-in-one commerce subscription.

The technology can be sophisticated while the operating cost remains controlled.

That is the fundamental concept behind this architecture.

29. The Larger Business Lesson

The most important lesson is not that Medusa is better than Shopify.

It is not that Railway is better than another cloud provider.

It is not that Astro is inherently better than another storefront framework.

The lesson is architectural:

Technology should be selected according to the economics and operating requirements of the business.

For a small eCommerce business, that means asking:

What must be reliable?

Payments.

Orders.

Inventory.

Customer information.

Shipping.

What should be automated?

Repetitive operational work.

What should be usage-based?

Payments, shipping, email, storage, bandwidth, and other activity-driven services where appropriate.

What should remain under business control?

Commerce data, storefront experience, business logic, integrations, and the ability to change providers.

What should not be built unnecessarily?

Infrastructure that specialized providers can provide more economically and reliably.

That is the philosophy behind the Warm Presents architecture.

Conclusion

The Warm Presents platform was designed around a simple business requirement: **keep the cost of operating an online store appropriate for a small business without sacrificing the capabilities customers expect from modern eCommerce.**

The resulting architecture does not accomplish that by stripping functionality away.

Instead, it decomposes the problem.

The storefront is optimized for customer experience.

Medusa manages commerce.

PostgreSQL manages durable business data.

Redis coordinates caching, events, workflows, and protection mechanisms.

Stripe manages payments.

Shippo manages shipping infrastructure.

Resend manages transactional communication.

Netlify and Railway provide managed infrastructure without requiring the business to operate physical servers.

Supabase Storage handles product media.

The components communicate through defined interfaces rather than being locked into a single monolithic platform.

The result is an architecture that attempts to align technology expense with business activity while simultaneously reducing the amount of repetitive work required from the owner.

That distinction is critical.

The goal was never simply:

“Build an inexpensive website.”

The goal was:

“Build an eCommerce business system whose operating economics make sense for a small business.”

That is a much more useful definition of cost-conscious engineering.

Appendix A — Current Architecture at a Glance

Layer	Technology	Business Responsibility
Customer storefront	Astro	Product presentation and customer shopping experience
Commerce platform	Medusa v2	Products, inventory, carts, orders, customers, authentication
Database	PostgreSQL	Durable commerce data
Infrastructure state	Redis	Cache, events, workflows, rate limiting, login protection
Backend hosting	Railway	Application, database, Redis infrastructure
Storefront hosting	Netlify	Customer-facing site delivery and builds
Object storage	Supabase Storage	Product images and uploaded files
Payments	Stripe	Payment processing
Shipping	Shippo	Rates, labels, tracking, address validation
Email	Resend	Transactional customer communication

The core backend architecture and integrations are documented in the project's build specification.

Appendix B — Current Known Limitations

The platform has moved beyond several of the implementation limitations identified during the earlier development phase.

Completed / No Longer Open

- **Product weights and dimensions are being populated.** The shipping system already supports product-specific shipment data, so adding accurate weights and dimensions

improves the accuracy of shipping quotes without requiring a change to the underlying shipping architecture.

- **Stripe is now operating with live production credentials.** Payment processing is no longer dependent on test-mode configuration.
- **Shippo is now operating with live production credentials.** Live shipping rates, label purchasing, and tracking are now part of the production operating environment.
- **Persistent tracking history is maintained per order.** Shipment tracking information is retained as part of the order record rather than relying solely on customer email notifications as the historical record.

Remaining / Deliberately Deferred

The following items remain either intentionally deferred or outside the current scope:

- **UPS and FedEx:** These carriers are not currently connected through Shippo. USPS remains the currently supported carrier integration.
- **Return labels:** Automated return-label creation remains intentionally unsupported at this stage.
- **Content Security Policy (CSP):** A comprehensive CSP remains a separate security-hardening task because it requires allow-listing the external services used by the storefront, including Stripe's Payment Element, the Netlify image infrastructure, and Google Fonts.
- **Repository-level CI/CD:** The project does not currently have a repository-level continuous integration/deployment pipeline. Testing and deployment verification remain operational processes rather than fully automated CI/CD workflows.

These items should be understood in context. They represent either **future enhancements**, **deliberate scope decisions**, or **operational maturity improvements**, rather than fundamental deficiencies in the commerce architecture.

The core production commerce functions—including payments, shipping, fulfillment, tracking, transactional email, customer authorization, rate limiting, and storefront synchronization—are operational components of the platform.

Appendix C — Sources and Research Notes

Primary project source

- Warm Presents Medusa Backend build specification, supplied for this analysis.

Architecture and platform research

- Medusa architecture and infrastructure-module documentation.
- Medusa workflow-engine documentation.
- Astro Islands Architecture documentation.
- Railway current pricing documentation.
- Netlify current pricing documentation.
- Netlify build-hook documentation.
- Supabase storage and billing documentation.
- Stripe current U.S. standard pricing.
- Shippo current API pricing.
- Resend current transactional-email pricing.

Pricing note: Provider pricing is time-sensitive. The figures referenced above reflect publicly published pricing retrieved in September 2026 and should be treated as research snapshots rather than guaranteed future pricing. The Warm Presents figure of about \$9 per month reflects the project's current Railway and Netlify plans and will change with plan selection and usage.